# Konzeption einer deklarativen und zustandsorientierten Sprache zur formalen Beschreibung und Simulation von Warteschlangen- und Transport-Modellen

Peter Eschenbacher

Juni 1993

Habilitationsschrift

erschienen bei:
SCS - Society for Computer Simulation Int.
Frontiers in Simulation Band 1
ISBN 1-56555-047-1

# Vorwort

Die vorliegende Arbeit entstand als Habilitationsschrift am Lehrstuhl für Betriebssysteme (IMMD 4) der Universität Erlangen-Nürnberg.

Sie behandelt die theoretischen Überlegungen, die der Modellbeschreibungssprache SIMPLEX-MDL zugrunde liegen. Diese Sprache ist Bestandteil des Simulationssystems SIMPLEX II, dessen Entwicklung im Rahmen des Forschungsvorhabens PAP (Projekt für flexibel automatisierte Produktionssysteme) in den Jahren 1985-1992 von der Firma Siemens gefördert wurde.

Ziel dieses Projekts war es, eine Arbeitsumgebung zu schaffen, mit deren Hilfe die Simulation von (geplanten) Produktionssystemen wesentlich erleichtert werden sollte. Nach Möglichkeit sollte der fertigungstechnische Planer Simulationsstudien ohne fremde Hilfe durchführen können.

Die zu Projektbeginn verfügbare Software machte eine mehr oder weniger aufwendige Programmierung der Modelle erforderlich, wofür spezielles Personal eingesetzt werden mußte.

Um diesem Mißstand abzuhelfen, entstanden seit dieser Zeit viele Simulationprogramme, die es erlauben, Modelle aus vorgefertigten Grundbausteinen mit Hilfe einer graphischen Bedienoberfläche zusammenzusetzen. Dadurch ergibt sich aber ein neues Problem: Vielfach reichen die vorhandenen Grundbausteine nicht aus und es müssen neue Bausteine programmiert und in das Grafiksystem eingepaßt werden. Der Aufwand für eine Simulationsstudie wird daher im Einzelfall sogar noch höher als zuvor.

Trotz gewisser Vereinfachungen in Bezug auf die Bedienung bleibt weiterhin die Notwendigkeit, Modelle oder Teile von Modellen leichter (und nach Möglichkeit auch für den Anwender) formal-sprachlich beschreibbar zu machen, um daraus ablauffähigen Code zu erzeugen. In dieser Arbeit werden daher die Möglichkeiten, um Warteschlangen-Modelle und Modellkomponenten mit den Mitteln einer Sprache zu formulieren, aus einer neuen, zustands-orientierten Sichtweise nochmals grundlegend durchdacht. Den Anstoß dazu gab die Tatsache, daß keine der damals bestehenden Simulationssprachen eine Modularisierung diskreter Modelle zuließ, andererseits aber die Systemtheorie für kontinuierliche Modelle aufzeigt, wie Modellzerlegung in idealer Weise möglich ist.

Es entwickelte sich die Zielvorstellung, die Vorteile des systemtheoretischen Konzepts auf diskrete Modelle und insbesondere Warteschlangenmodelle zu übertragen.

Der Weg dorthin erwies sich jedoch als mühsam und langwierig. Vor allen Dingen stellte sich heraus, daß die ursprünglichen Mittel der Systemtheorie zwar prinzipiell die Beschreibung von Warteschlangenmodellen ermöglichen, aber nicht wie erhofft eine modulare Modellzerlegung gestatten. Aus diesem Grunde mußte die theoretische und methodologische Grundlage für die Beschreibung und Simulation von Warteschlangenmodellen erst durch eine sinnvolle Ergänzung des systemtheoretischen Paradigmas geschaffen werden.

Auch zeigte sich, daß die zustandsorientierte Sichtweise der Systemtheorie mit den Modellierungsansätzen der herkömmlichen und prozedural orientierten Simulationssprachen keinerlei Gemeinsamkeit oder Ähnlichkeit aufwies. Im Laufe der Zeit wurde immer deutlicher, daß die angestrebte Sprache deklarativen Charakter besitzt und damit eine ganz andere Denkweise voraussetzt, als man es von prozeduralen Sprachen gewohnt ist.

In meiner Dissertation [Esch 90] wurde zunächst untersucht, wie sich die Beschreibung von kontinuierlichen und von diskreten Vorgängen miteinander in Einklang bringen läßt. In dieser Arbeit werden nun diejenigen Erweiterungen des Sprachkonzepts beschrieben, die eine Formulierung von Warteschlangen- und Transportmodellen möglich machen. Eine Kenntnis der Dissertation ist dazu nicht vonnöten, da alle für das Verständnis erforderliche Grundlagen auch in dieser Abhandlung enthalten sind.

Im Mittelpunkt der Ausführungen stehen die Eigenarten der deklarativen Beschreibungsform. Die deklarative Beschreibung verlangt nach Vollständigkeit und Widerspruchsfreiheit. Diese beiden Kriterien, deren Einhaltung nach Möglichkeit bereits vom Compiler garantiert werden soll, prägen neben der Forderung nach modularer Modellzerlegung ganz entscheidend die Ausgestaltung der Sprache.

Für die Beherrschung umfangreicher Modelle, für die Wiederverwendbarkeit von (Teil-) Modellen sowie für eine Zusammenarbeit mehrerer Personen am gleichen Projekt ist die Modularisierbarkeit wesentlich und unverzichtbar. Es zeigt sich jedoch, daß diese Forderung im Konflikt mit anderen Gesichtspunkten (Eindeutigkeit und Autonomie) steht, und daher sehr sorgfältig eine Kompromißlösung ausgearbeitet werden muß.

Die syntaktische Notation der entworfenen Sprache wird vollständig dargestellt. Die theoretischen Ausführungen werden für den Leser dadurch griffiger und die Beispiele sind in ihrer Ganzheit zu verstehen zu verstehen. Es wird sich auch herausstellen, daß eine geeignete Syntax einzelne Anforderungen an die Sprache unterstützen kann.
Theoretische Betrachtungen neigen dazu, möglichst allgemeine Formulierungen zuzulassen. Abgesehen davon, daß komplexe Formulierungen wegen ihrer Unübersichtlichkeit selten genutzt werden, wird die Implementierung sehr umfangreich und ineffizient in der Ausführung. Daher werden geeignete Einschränkungen diskutiert, welche die Anwendungsbreite der Sprache nicht mindern, aber viel zur Einfachheit und Effizienz beitragen.

Auf dem Gebiet der Simulation gibt es bislang kaum Betrachtungen über das Thema Modellspezifikation, auf die man Bezug nehmen oder auf die man verweisen könnte. Um die Gedankengänge nachvollziehbar zu machen, wurde die Materie streckenweise – für wissenschaftliche Abhandlungen unüblich – induktiv statt deduktiv dargelegt. Als Grundlage für weitere Arbeiten an diesem Thema erschien mir diese Vorgehensweise die geeignetere.

Ich denke, daß die Überlegungen, die in dieser Arbeit angestellt werden, auch über die Bewältigung von Simulationsaufgaben hinaus von Interesse sind. Vielfach handelt es sich um Probleme, die sicherlich in ähnlicher Form bei der Beschreibung anderer Sachverhalte auftreten. Auch das Sprachkonzept selbst soll Anregungen für einen Sprachentwurf in anderen Disziplinen der Informatik liefern. Vor allem auf den Bereich der Steuerungstechnik und Prozeßautomatisierung scheint das Konzept übertragbar.

Die Arbeit wäre in dieser Form sicherlich nicht zustande gekommen ohne die Mitwirkung meiner beiden Kollegen, Herrn Jochen Wittmann und Herrn Thomas Apsel. Ihnen gebührt ein herzliches Dankeschön für die vielen Beiträge, die sie in zahlreichen Diskussionen beigesteuert haben.

Ebenso geht mein Dank an Herrn Prof. B. Schmidt, der diese Arbeit angeregt und betreut hat, an Herrn Prof. F. Hofmann, der mir stets mit Tips und Ratschlgen zur Seite stand, und Herrn Prof. H. Wedekind sowie Herrn Prof. W. Ameling, die zusammen mit den beiden erstgenannten die Arbeit begutachteten.

Erlangen, im Februar 1995 P. Eschenbacher

# Preface

This book has been submitted for the certificate of habilitation at the chair for operating systems (IMMD 4) at the university of Erlangen-Nuremberg.

It deals with the theoretical considerations which underlie the model description language SIMPLEX-MDL. This language ist part of the simulation system SIMPLEX II, the development of which was sponsered by the Siemens company from 1985-1992, in connection with the research project PAP (project for flexibly automated production systems).

It was the aim of this project to build a working environment, which makes simulation of (planned) production systems much easier. If possible, the planner of the production system should be able to perform simulation studies without assistance.

The software available at the beginning of the project required great efforts to program these models, and specially skilled personnel were needed.

Since that time, many simulation programs have been developed to improve this situation, which enable a model composition with pre-defined components to be constructed with the aid of a graphical user interface.
However, this has given rise to a new problem: Often the existing components are not sufficient and new modules must be programmed and integrated into the graphic system. In certain cases the costs for a simulation study are even higher than before.

In spite of many simplifications concerning usage, the necessity has remained to make it easier (if possible for the user) to describe models and parts of models in a formal language in order to generate executable code from it.
In this book, based on a new, state-oriented viewpoint, the possibilites of formulating queuing systems and system components by means of a formal language are revisited and thoroughly discussed.
This was initiated by the fact that none of the former simulation languages enabled the modular decomposition of discrete models, but from a different viewpoint, system theory demonstrates for continuous models how modularization can be practiced in an ideal way.

A target was set up, to transfer the advatages of the system theoretic concept to discrete models, especially those involving queuing models.

But that way turned out to be difficult an lengthy. Above all, it became apparent that the original means of system theory enable a description of queuing systems in principle, but not a modular model decomposition as hoped. For this reason at first the theoretic and methodologic foundation to describe and simulate queuing systems had to be created as a useful supplement to the system theoretic paradigm.

Furthermore it turned out that the state-oriented viewpoint of system theory and the traditionally procedural simulation languages have no similarities. In the course of time the character of the desired language appeared more and more to be declarative. This requires quite a different way of thinking from that used with procedural languages.

My dissertation [Esch 90] deals with the problem how the description of continuous and discrete processes can be unified. Here in this book those supplements of the language concept are described, which make a formulation of queuing and transport systems possible. Knowledge of the dissertation is not required, for everything that is needed will be found in this book as well.

In the center of the explanations there are the peculiarities of describing in a declarative way. The declarative description claims to be complete and free of contradiction. These two criteria, which should be observed and controlled by the compiler, together with the demand for modular model decomposition, determine the language design considerably.

To master large models, to re-use parts of models and to work at the same project in a team model decomposition is important and cannot be relinquished. However it must be pointed out that this claim conflicts with other aspects (definiteness and autonomy) and therefore a compromise solution carefully must be worked out.

The syntactic notation of the designed language is completely explained. The theoretic explanations become clearer to the reader and the examples can be wholly understood. It turns out that a proper syntax can support some of the requirements given to the language.

Theoretic studies tend to admit general formulations as far as imaginable. Despite of the fact that complex formulations are seldom used, the implementation becomes voluminous and inefficient during execution. Therefore useful limitations are discussed, which do not deminish the field of application but lead to much simplicity and efficiency.

So far, in the field of simulation there are few discussions concerning the topic of model specification which can be referenced. To be able to better follow the chain of reasoning, the material was explained in an inductive rather than deductive way, which is not very common in scientific dissertations. For a better understanding, and as a basis for further studies of the topic, this procedure seems to be the more appropriate.

I believe that the considerations presented in this book are of interest beyond accomplishing the solution of simulation problems. In many cases they deal with problems which certainly occur in a similar way when other aspects of reality have to be described. The concept of the language itsself could also give impulse to language design in other disciplines within computer science. This concept seems to be transferable, especially in the fields of control technique und process automation.

This book certainly did not appear in this form without the contributions of my two colleagues, Jochen Wittmann and Thomas Apsel. A cordial thankyou to both of them.

Just as much I thank Prof. B. Schmidt, who initiated this work, Prof. F. Hofmann, who contributed many tips and advices, as well as Prof. H. Wedekind and Prof. W. Ameling, who evaluated this book together with those named above.

Erlangen, February 1995 P. Eschenbacher

# Préface

Ce livre a été soumis afin d'obtenir le certificat d'ingénieur, pour une chaire de systèmes opérationnels (IMMD4) de l'Université de Erlangen a Nurembourg.

Ce livre traite des considérations relatives au langage de modélisation SIMPLEX-MDL. Ce langage fait partie du système de simulation SIMPLEX II, dont le développement a été soutenu dans le cadre du projet de recherche PAP (projet relatif aux systèmes de productions flexibles automatisés) durant les années 1985-1992 chez Siemens.

Le but de ce projet était de construire un environnement de travail, qui facilite la simulation de systèmes de production en phase de conception. Si possible, un concepteur non spécialiste devrait être en mesure de faire la simulation sans aucune aide extérieure. Le logiciel présent au début du projet exigeait d'importants efforts de programmation des modèles, ce qui nécessitait du personnel spécialisé.

A l'heure actuelle, cette situation s'est améliorée grâce à l'utilisation d'interfaces graphiques facilitant le développement des modèles et l'utilisation de parties pré-définies. Cependant, quand un nouveau problème se présente, les parties existantes ne sont pas suffisantes. De nouveaux modèles devraient être programmés et intégrés dans le système graphique. Dans certains cas, les dépenses pour l'étude de la simulation se montrent encore plus élevées qu'auparavant.

En dépit des simplifications d'utilisation, il est nécessaire (si possible pour l'utilisateur) de décrire des parties de modèles dans un langage formel, afin de générer du code utilisable. En se basant sur une nouvelle approche, les possibilités de description des systèmes d'attente et des parties de systèmes à l'aide d'un langage formel seront discutées. Ceci a été justifié, dans ce livre, par le fait qu' aucun des langages de simulation existants permettaient la décomposition modulaire des modèles discrets, mais aussi par le fait que la théorie des systèmes montrait que pour les modèles continus, la modélisation pouvait être pratiquée plus simplement. L'objectif envisagé était de transférer les avantages d'une idée théorique sur des modèles discrets, surtout les modèles d'attente. Mais cette voie se montra longue et difficile. Surtout, il était évident qu'en principe les moyens originaux des systèmes théoriques mettent en mesure la description des systèmes d'attente, mais ne permettent pas une approche modulaire de décomposition comme il avait été espéré. Pour cette raison, il était nécessaire de s'intéresser d'abord à la fondation théorique et méthodologique pour décrire et simuler les systèmes d'attente en utilisant l'apport d'un paradigme théorique des systèmes.

En plus, il devenait apparent que le point de vue était orienté sur les langages de simulation traditionnel, et non pas sur les simulations. Au cours du temps, le caractère du langage désiré apparaissait de plus en plus déclaratif et ceci nécessitait une toute autre maniere de penser, comme l'utilisation de langages procéduraux.

Ma thèse [Esch 90] traite de ce problème : "comment améliorer la description des processus continus et discrets". Içi, les suppléments de ce langage sont décrits, ce qui rend la formulation des systèmes de transport et d'attente possible. La pré-connaisance de la thèse n'est pas nécessaire, parce que tout ce qui est nécessaire sera traité dans ce livre.

Au travers des explications, le lecteur trouvera les particularités de description des systèmes d'une maniere déclarative. La description déclarative prétend être complète et sans contradiction. Ces deux critères, qui devraient être observés et controlés par un compilateur, avec

une décomposition modulaire du modèle. Ceçi détermine considérablement la construction du modèle.

Afin, de maitriser les modèles complexes, de réutiliser certaines parties des modèles et de travailler en équipe, il est important de voir comment la décomposition des modèles est effectuée sans perdre les modèles originaux. Néanmoins, j'indiquerai que cette prétention se met en conflit avec d'autres aspects (l'aspect définitif et autonome) et pour cette raison une solution de compromis doit être trouvée.

La notation synthétique du langage décrit est complètement expliquée. Les explications théoriques deviennent plus claires aux lecteurs et les exemples peuvent être totalement compris. Il devient apparent qu'une syntaxe propre peut supporter certaines des exigences du langage. Les études théoriques ont tendance à admettre des formulations généralement imaginables. En dépit du fait que les formules complexes sont peu utilisées, l'implémentation devient volumineuse et l'éfficacité diminue pendant l'exécution. Pour cette raison, des limitations utiles sont discutées, qui ne diminuent pas les applications mais qui conduisent à plus de simplicité et d'éfficacité.

Jusqu'à présent, dans la simulation, il n'y a que peu de considérations à propos des spécifications des modèles qui peuvent être approuvées. Afin de mieux suivre le cours des idées, on a donné des explications sur le materiel d'une manière inductive et non déductive, ce qui n'est pas toujours commun dans des thèses scientifiques. Pour une meilleure compréhension et comme base pour des études plus avancées à propos de ce projet, ce livre me semble plus opportun.

Je crois, que les considérations presentées dans ce livre seront intéressantes au delà des aspects classiques des problèmes de simulation. Dans beaucoup de cas, elles traitent des problèmes qui se manifestent d'une manière identique quand d'autres aspects de la réalité doivent être décrits. Le concept d'un langage même pourrait aussi donner une impulsion au développement de langages dans d'autres disciplines informatiques. Ce serait surtout valable dans les techniques de contrle et les processus d'automatisation.

Ce livre ne serait certainement pas paru sous cette forme sans la contribution de deux de mes collègues, Jochen Wittmann et Thomas Apsel. Aussi, je les en remercie infiniment.

Je remercie, aussi le Professeur B. Schmidt qui a débute ce travail et le professeur F. Hofmann qui m'a prodigué de précieux conseils et les professeurs H. Wedekind et W. Ameling qui ont effectué l'évaluation de ce livre en coopération avec les susmentionnés.

Erlangen, Février 1995 P.Eschenbacher

# Inhaltsverzeichnis

# Kapitel 1

# Einleitung

Die vorliegende Arbeit stellt konzeptionelle Überlegungen für eine deklarative Sprache zur Beschreibung von dynamischen Modellen (Modellbeschreibungssprache) vor. Der Schwerpunkt liegt dabei auf der Formulierung von Warteschlangen- und Transportmodellen. Es wird jedoch Wert darauf gelegt, daß die Verwandtschaft zu entsprechenden kontinuierlichen Modellen deutlich wird, um ein für diskrete und kontinuierliche Modelle einheitliches Konzept zu schaffen.

Die Abhandlung ist von zwei Seiten motiviert. Da ist zum einen das Verlangen der Fertigungstechnik, mit einem auf den Planer zugeschnittenen Simulationssystem neuartige Produktionssysteme detailgenau modellieren, simulieren und analysieren zu können. Und zum anderen ist da das Bedürfnis des Informatikers, das Thema Simulationssprachen, das etwas in Vergessenheit geriet, mit neueren sprachlichen Konzepten zu bereichern. Da sich die Anforderungen der Fertigungstechnik mit den herkömmlichen Simulationssprachen ohnehin nicht abdecken ließen, bot sich die Gelegenheit, bei der Konzeption einer neuen Sprache beide Zielsetzungen gemeinsam zu verfolgen und damit eine Synthese praktischer Notwendigkeiten mit theoretischen Überlegungen zu versuchen.

In dieser Einleitung wird zunächst die Thematik dieser Abhandlung aus dem übrigen Bereich der Simulationstechnik ausgegrenzt. Daraufhin wird gezeigt, welche Aufgaben ein Simulationsprogramm im allgemeinen zu erfüllen hat und mit welchen Arten von Werkzeugen Simulationsprogramme erstellt werden können. In Abgrenzung von den traditionellen Simulationssprachen werden die grundlegenden Gesichtspunkte einer Modellbeschreibungssprache herausgestellt und damit der hier eingeschlagene Weg umrissen.

Schließlich wird am Beispiel der Fertigungstechnik erläutert, welche Anforderungen an die Simulationstechnik und insbesondere an Werkzeuge zur Modellerstellung heute gestellt werden. Unterschiedliche Ansätze, diese Anforderungen zu erfüllen werden diskutiert.

Im zweiten Kapitel wird die systemtheoretische Modelldarstellung als neues Programmiermodell grundgelegt und die elementaren Sprachkonstrukte einer Modellbeschreibungssprache eingeführt. Das dritte Kapitel erörtert die Probleme dieses Paradigmas bei der Modellierung von Warteschlangen- und Transportmodellen und versucht, diese durch Ergänzungen der Sprache und des Paradigmas zu beheben. Im vierten Kapitel wird diskutiert, auf welche Weise Transportvorgänge – insbesondere bei modularer Modellzerlegung – beschrieben werden können. Das fünfte Kapitel faßt den Sprachentwurf zusammen und stellt ihn systematisch dar. Das sechste Kapitel spricht Spracherweiterungen an, die erforderlich sind, um die Sprache für praktische Anwendungen einsetzen zu können. Schließlich werden im siebten Kapitel

anhand von vollständigen Beispielen die Einsatzmöglichkeiten der Sprache aufgezeigt. In der Zusammenfassung wird zuletzt ein Resumee gezogen und geschildert, inwieweit die gesetzten Ziele erreicht werden konnten.

## 1.1 Aufgabenfelder der Simulationstechnik

Simulieren heißt Experimentieren mit Modellen. An die Stelle eines realen oder fiktiven Systems (originales System) tritt ein Simulationsmodell. Mit diesem werden Experimente durchgeführt und man hofft auf Ergebnisse, die denen des originalen Systems entsprechen.

Simulation auf Digitalrechnern bedeutet, daß mit Hilfe eines auf einem Rechner ablaufenden Simulationsprogramms Experimente durchgeführt und Ergebnisse gewonnen werden. Das Simulationsprogramm basiert auf einer (gedanklichen) Modellvorstellung vom originalen System, die in mathematisierter Form, in graphischer Form oder auch in rein informeller, verbaler Form vorliegen kann.

Die Modellvorstellung erhält man durch Analyse von Beobachtungen (Messungen), die am realen Systems vorgenommen wurden, oder durch die Spezifikation eines erdachten (fiktiven), technischen Systems.

Die Ergebnisse von Simulationsexperimenten liegen zunächst als Zeitreihen (ggf. auch statistisch komprimiert) auf Dateien vor und müssen, will man sie interpretieren oder mit anderen Ergebnissen vergleichen, aufbereitet und visualisiert werden.

Diesen Aufgaben entsprechend gliedert sich die Simulationstechnik in die beiden Kernbereiche

- Erstellen des Modells (d.h. des Simulationsprogramms)
- Experimentieren mit dem Modell

und die beiden angrenzenden Bereiche

- Modellbildung (Systemanalyse) und
- Aufbereitung und Visualisierung der Ergebnisse,

welche auch Bestandteil anderer Disziplinen sind.

Im Verlauf der letzten 30 Jahre hat die Simulation auf Digitalrechnern die Simulation mit anderen physikalischen Systemen weitgehend verdrängt, da der Digtalrechner alle Bereiche wirksam unterstützt und mittlerweile für viele Anwendungen auch die erforderliche Rechengeschwindigkeit besitzt.

Von 1985-1991 wurde am Institut für Mathematische Maschinen und Datenverarbeitung (Lehrstuhl für Betriebssysteme) das Simulationssystem SIMPLEX II [Lang 89, Esch 90, Dörn 91] entwickelt, das eine weitgehende Unterstützung in allen Bereichen bietet und alle dazu erforderlichen Funktionen unter einer gemeinsamen Bedienoberfläche vereint. Diese Abhandlung stützt sich zum großen Teil auf Erkenntnisse und Erfahrungen, die bei der Entwicklung und Anwendung des Simulationssystems SIMPLEX II gesammelt werden konnten.

Wir beschränken uns in den Ausführungen hier allerdings auf den Bereich der Modellerstellung. Ziel ist es, einerseits Modellvorstellungen formal beschreiben zu können, andererseits aus dieser formalen Beschreibung durch einen Compiler ein Simulationsprogramm generieren (konstruieren) zu lassen.

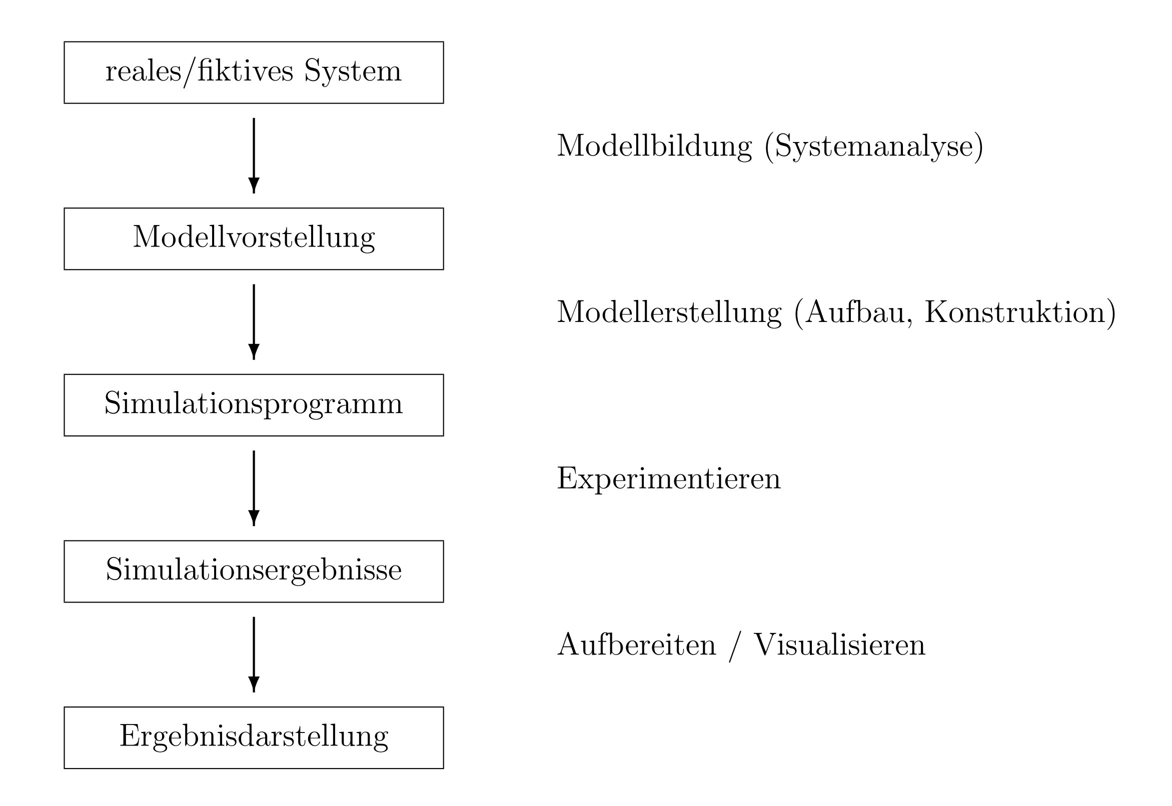



Abb. 1.1-1: Aufgabenfelder der Simulationstechnik

## 1.2 Aufgaben eines Simulationsprogramms

Jedes Simulationsprogramm besteht aus dem Code, der das Modell simuliert, und dem Code, der die Programmschnittstellen herstellt.

Ein Simulationsprogramm kann zu unterschiedlichen Zwecken eingesetzt werden. Die Schnittstellen zur Umgebung, welche die Einsatzmöglichkeiten eines Programms bestimmen, sind jedoch in ihrer Funktionalität stets gleich.

Man setzt Simulationsprogramme

- zur Durchführung von Simulationsstudien oder
- zu Übungs- und Trainingszwecken

ein.

Bei der Durchführung von Simulationsstudien will man Erkenntnisse über das modellierte System gewinnen. Hierzu führt man eine Vielzahl einzelner Simulationsläufe mit immer wieder neuen Anfangszuständen und Parametrierungen aus, wertet anschließend die Ergebnisse aus und läßt sie graphisch darstellen. In den meisten Fällen liegt der Zweck von Simulationsstudien im Auffinden optimaler Parameter, Strategien und Modellstrukturen.

Beim Einsatz eines Simulators zu Übungs- und Trainingszwecken will man den Bediener im Umgang mit dem modellierten System schulen. Während des Simulationslaufs wird das Verhalten des Modells fortwährend ausgegeben und der Bediener reagiert auf diese Ausgaben mit ständig neuen Eingaben an das Modell. Zu diesem Einsatzfeld zählen u.a. Planspiele oder Fahrzeugsimulatoren.

Während man im ersten Fall mit einem abgeschlossenen Modell experimentiert, ist im zweiten Fall das Modell für laufende Eingaben offen.

Für Simulationsstudien ist das Simulationsprogramm mit Eingabe-Schnittstellen zu versehen, die

- eine Belegung des Anfangszustands (einschließlich der Parameter)
- ein Definieren der Beobachter sowie
- ein Setzen der Steuerparameter

erlauben.

Beobachter ermöglichen es, den Verlauf von Variablen in Abhängigkeit von der Zeit (Zeitreihe) oder in Abhängigkeit vom Eintreten bestimmter Ereignisse (Ereignisreihe) aufzuzeichnen. Die Aufzeichnung kann (bei diskreten Variablen) vollständig oder komprimiert erfolgen.

Dementsprechend wird ein Beobachter durch die Art der Aufzeichnung, durch einige Parameter und durch eine Liste von Variablen festgelegt, die beobachtet werden sollen. Die Parameter besagen, zu welchen Zeitpunkten bzw. bei welchen Ereignissen die Variablen beobachtet und (bei einer Komprimierung) zu welchen Zeitpunkten sie aufgezeichnet werden sollen.

Die Komprimierung eines Zeitverlaufs kann dadurch erfolgen, daß

- die Variablenwerte nur in bestimmten Zeitabständen aufgezeichnet werden
- Minimum, Maximum und Mittelwert der Verlaufsfunktion am Ende von periodischen Beobachtungsintervallen aufgezeichnet wird
- die (zeitlich gewichtete) Verteilung der Verlaufsfunktion am Ende des Simulationslaufs aufgezeichnet wird.

Die Steuerparameter dienen der Auswahl und der Parametrierung der verwendeten Algorithmen. Hierzu zählen beispielsweise Zufallgeneratoren und deren Startwerte.

Damit der Anwender zügig und übersichtlich seine Experimente durchführen kann, sind diese Eingabe-Schnittstellen komfortabel zu gestalten. Insbesondere muß es möglich sein, die Modellvariablen symbolisch zu benennen. Wie die Arbeiten [Lang 89] und [Ulb 92] zeigen, benötigt man hierzu jedoch nur die Symbolinformation auf Datei. Eine Codegenerierung zur Einrichtung dieser Schnittstellen ist nicht erforderlich.

Die Ausgaben eines Simulationsprogramms umfassen

- den nach Ablauf des Simulationslaufs erreichten Endezustand mit sämtlichen Modellvariablen
- die Aufzeichnungen der Beobachter
- ggf. ein Ablaufprotokoll und eine Ablaufstatistik.

Ablaufprotokoll und Ablaufstatistik geben Aufschluß über die ausgeführten Aktionen des Simulationsprogramms und deren Anzahl und sind eine Hilfe beim Test des Simulationsmodells.

Endezustand und Aufzeichnungen der Beobachter sind in einem offengelegten Datenformat abzuspeichern. Auf diese Weise kann die Nachbereitung sowohl durch das Simulationssystem als auch durch fremde (benutzereigene) Auswerte- und Darstellungsprogramme (evtl. nach einer Konvertierung des Datenformats) vorgenommen werden.

Um bei einem Trainingssimulator das Verhalten eines Modells im laufenden Betrieb studieren zu können, ist – neben den bereits genannten Schnittstellen – eine Kommunikation mit externen Programmen (z.B. über TCP/IP) vorzusehen. Diese Kommunikation dient einerseits der fortlaufenden Ausgabe von Beobachtungen, andererseits der fortlaufenden Eingabe von Geschehnissen aus der Umgebung (z.B. das Drücken eines Gaspedals), welche auf die Dynamik des Modells Einfluß nehmen. Für die Ein- und Ausgabe muß das Modell geeignete Variablen bereitstellen. Auch für diese Schnittstelle ist keine individuelle Codegenerierung, sondern nur Symbolinformation auf Datei erforderlich.

Ein Echtzeitbetrieb oder ein echtzeitproportionaler Betrieb als Trainingssimulator läßt sich herzustellen, indem der Simulator immer wieder aufs neue von außen angestoßen wird und stets nur bis zu einem vorgegebenen Zeitpunkt weiterläuft. Auch für eine derartige Triggerung ist eine geeignete Schnittstelle vorzusehen.

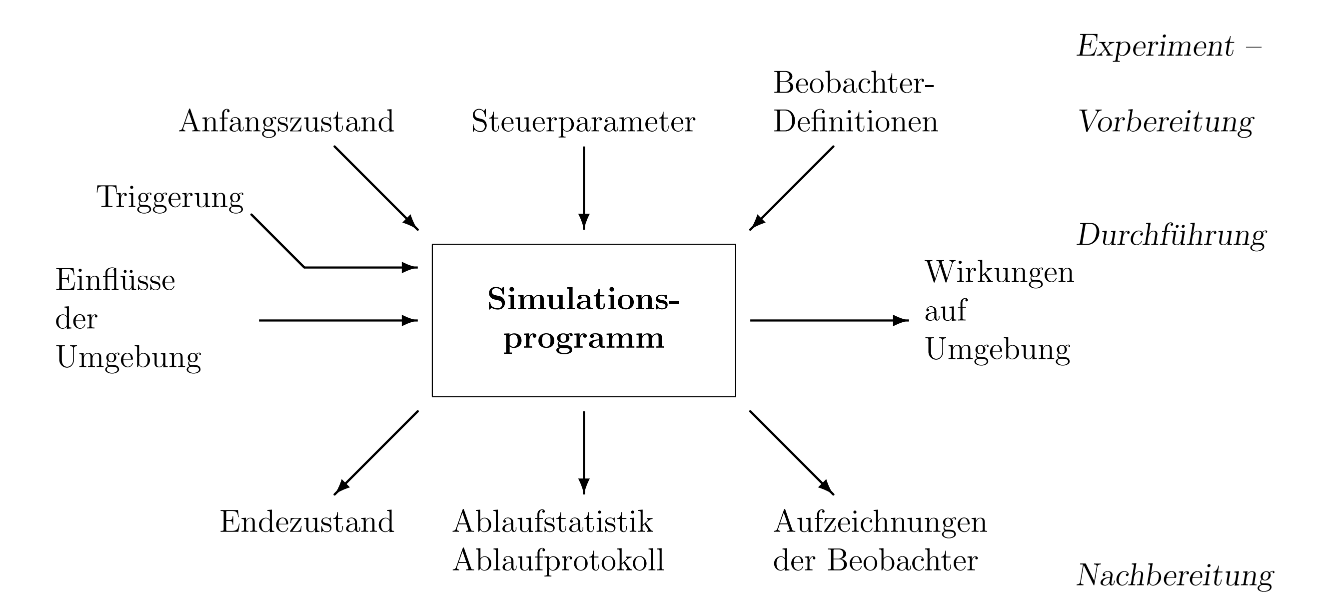



Abb. 1.2-1: Schnittstellen eines Simulationsprogramms

Wichtig ist es festzuhalten, daß alle Simulationsprogramme mit den gleichen Schnittstellen auskommen und diese Schnittstellen in gleicher Weise codiert sein können, da sie Symbolinformationen nur auf Datei benötigen. Dadurch braucht der Verwendungszweck eines Modells nicht bereits bei seiner Erstellung festgelegt werden und eine automatische Codegenerierung braucht keinen Code für die Schnittstellen liefern.

Lediglich der Code, der das Modell simuliert, ist individuell zu erzeugen. Auch dieser Code läßt sich wiederum in einen gleichbleibenden und einen individuellen Code trennen. Der individuelle Code repräsentiert das spezifische Modellverhalten, der gleichbleibene Code steuert die Ausführungsreihenfolge des modellspezifischen Codes (siehe hierzu Kapitel 2).

Läßt sich der Code in dieser Weise trennen, dann zerfällt ein Simulationsprogramm in drei Teile:

- Repräsentation des spezifischen Modellverhaltens
- Steuerung der Ausführungsreihenfolge (Ablaufsteuerung)
- komfortable Ein-/Ausgabeschnittstellen

Von diesen drei Teilen sollte sich der Anwender nach Möglichkeit nur mit der Repräsentation des Modells beschäftigen müssen. Die beiden anderen Aufgaben sollten ihm durch den Einsatz geeigneter Software-Werkzeuge vollständig abgenommen werden.

In einem prozeduralen Simulationsprogramm läßt sich jedoch das Modellverhalten nicht ohne Kenntnis der Ablaufsteuerung und ohne Kenntnis der Datenstrukturen repräsentieren. Es wird daher das Ziel sein, eine formale Beschreibung für das Modellverhalten zu finden, die keine Kenntnis der Ablaufsteuerung benötigt. Hingegen ist die Repräsentation des Modellverhaltens von den Schnittstellen eines Simulationsprogramms unabhängig und deshalb werden wir uns mit der Realisierung der Schnittstellen nicht weiter auseinandersetzen.

## 1.3 Erstellung von Simulationsprogrammen

In [Page 88] werden die Programmiersprachen Modula-2, C und Ada sehr ausführlich auf ihre Eignung für diskrete Simulationsprogramme untersucht. Hierbei werden zwei prinzipiell verschiedene Implementierungskonzepte unterschieden:

– ereignis-orientierte Simulation
– prozeß-orientierte Simulation

Während ereignis-orientierte Simulationsprogramme nur einen einzigen Programmfluß enthalten und über einen programmeigenen Scheduler zu den einzelnen Aktivitäten verzweigt wird, beschreiben prozeß-orientierte Simulationsprogramme parallele Abläufe, die sich gegenseitig aktivieren oder nach einer bestimmten Ruhezeit selbst wieder aktiv werden.

Bei gleicher Modellvorstellung führen die beiden Konzepte zu völlig verschieden Programmen.

Ereignisorientierte Simulation läßt sich mit jeder Programmiersprache hinreichend gut realisieren, prozeß-orientierte Simulation jedoch nur, wenn die Sprache Nebenläufigkeiten unterstützt und die erforderlichen Synchronisationsmechanismen (z.B. signal/wait) zur Verfügung stellt. Von den drei angeführten Programmiersprachen kommt hierfür nur ADA in Betracht.

Die Erstellung eines (prozeduralen) Simulationsprogramms unmittelbar aus einer (nicht formal dokumentierten) Modellvorstellung ist zwar möglich und wird leider auch heute noch in vielen Fällen praktiziert, das Resultat ist jedoch meist nicht zufriedenstellend. Hierfür gibt es eine Vielzahl von Gründen:

- Eine nicht formal dokumentierte Modellvorstellung ist in der Regel nicht präzise abgefaßt und meist unvollständig. Häufig liegt sie teilweise nicht einmal in schriftlicher Form vor. Dadurch sind dem Programmierer viele Freiheiten gegeben, nach eigenem Gutdünken die bestehenden Lücken auszufüllen. Da der Programmierer in der Regel wenig von der Anwendung versteht, enthalten Simulationsprogramme oft falsche Annahmen und Vermutungen.

- Das Simulationsprogramm, mit dem schließlich die Ergebnisse gewonnen werden, ist als Dokument für das Modell schlecht geeignet, da man auf das Modellverhalten nur schließen kann, wenn man sowohl die Anweisungen, welche Aktionen des Modells darstellen, als auch die Anweisungen, welche die Reihenfolge dieser Aktionen festlegen, analysiert.

- Größere Simulationsprogramme werden sehr schnell unübersichtlich, wenn man sich nicht strikt an einen geeigneten Programmierstil hält und sind daher keine Aufgabe für einen unerfahrenen Anwender. Der Grund liegt darin, daß ein Modell viele parallel ablaufende Aktivitäten beinhaltet, die in einer geeigneten Reihenfolge sequentiell abgearbeitet werden müssen. Hierdurch wird vor allem eine modulare Zerlegung des Programms zum Problem.

- Wegen der schlechten Lesbarkeit fällt es schwer, ein ohne Werkzeuge erstelltes Simulationsprogramm zu ändern, zu erweitern oder durch einen anderen Programmierer fortführen zu lassen. Die Weitergabe an einen Fachkollegen bringt diesem meist wenig Nutzen.

- Die Analyse einer Reihe von bestehenden Simulationsprogrammen ergab, daß diese häufig methodische Fehler enthalten und die Aktionen nicht in einer geeigneten Reihenfolge ausgeführt werden. Insbesondere ist für die Modellvariablen meist nicht klar festgelegt, ob der neu errechnete Wert zum aktuellen Zeitpunkt oder für den folgenden Zeitpunkt Gültigkeit besitzt. Die Reaktion des Programms auf gleichzeitig eintretende Ereignisse ist damit nicht klar bestimmt und so erzeugen Simulationsprogramme mit derartigen Fehlern immer wieder überraschende, nicht gewollte Ergebnisse.

- Modelluntersuchungen beginnt man meist mit kleinen Modellen und ist daher geneigt, diese auf die Schnelle zu Programmieren. Wird daraus allmählich eine ernsthafte Anwendung, dann nehmen die Programme sehr schnell einen nicht erwarteten Umfang an. Das liegt nicht nur daran, daß das Modell umfangreicher wird, sondern vor allem auch daran, daß immer mehr Komfort für das Experimentieren erforderlich wird, will man nicht das Programm für jedes Experiment neu übersetzen. Die Erstellung von Benutzerschnittstellen, die so gestaltet sind, daß der Anwender selbst mit dem Modell experimentieren kann, erfordern einen nochmals deutlich höheren Zeitaufwand.

- Der Test eines Simulationsprogramms ist sehr aufwendig, da sehr viele Situationen ausgetestet werden müssen. Dazu ist das Simulationsprogramm jedesmal gezielt in einen gewünschten Zustand zu bringen und die zu erwartenden Ergebnisse von Hand zu berechnen. Ohne die Möglichkeiten, beliebige Anfangsbelegungen vorzunehmen und ausführliche Protokolle ausgeben zu lassen, ist ein Test gar nicht praktikabel.

Zusammenfassend läßt sich feststellen, daß

a) zwischen der Gedankenwelt des Anwenders und des Programmierers eine tiefe Kluft besteht

b) sich die Erstellung eines Simulationsprogramms als sehr anspruchsvoll und aufwendig erweisen kann.

Der Programmierer versteht zu wenig von der Anwendung und der Anwender versteht nicht das Simulationsprogramm. Auf jeden Fall muß man davon ausgehen, daß der Anwender sein Modell nicht selbst programmieren kann.

Diese Problematik wurde schon sehr frühzeitig erkannt und man versuchte, durch verschiedene Hilfsmittel Verbesserungen zu erreichen.

Diese wollen wir unterscheiden in

a) simulations-unterstützende Programmiersprachen

b) Programmpakete für die Simulation
c) datengesteuerte Simulatoren
d) Simulationssprachen

## Simulations-unterstützende Programmiersprachen

Die simulations-unterstützenden Programmiersprachen zeichnen sich durch Spracherweiterungen aus, welche Simulationsanwendungen unterstützen sollen. Hierzu zählen SIMULA [Rohl 73] sowie alle objektorientierten Sprachen wie SMALLTALK [Kaehl 86], C++ [Strou 87] oder Object-C [Schnu 91].

Die Unterstützung beschränkt sich jedoch im wesentlichen auf die Bildung von abstrakten Datentypen, sogenannte Objekt-Klassen, von denen mehrfache Ausprägungen (Objekte) angelegt werden können. Jedes Objekt besteht aus einem individuellen Satz von Variablen und einem Satz von Funktionen (Methoden), die nur auf den Datenbeständen des Objekts operieren und allen Objekten einer Klasse gemein sind. Ein Objekt kann über Botschaften Methoden eines anderen Objekts anstoßen.

Nicht vorgesehen ist im allgemeinen jedoch die Nebenläufigkeit von Objekten. Eine Ausnahme davon bildet lediglich SIMULA, das damit prozeß-orientierte Modellimplementierungen erlaubt.

Das von den objekt-orientierten Sprachen angebotene Vererbungskonzept erscheint uns für Simulationsanwendungen weniger nutzbringend. Bei keiner, der von unserer Arbeitsgruppe zahlreich durchgeführten Anwendungen, hätte dieses Konzept nennenswerte Vorteile gebracht.

Letztendlich liefern objekt-orientierte Sprachen "nur" ein Konzept für die Modularisierung von Programmen. Richtig eingesetzt, werden Simulationsprogramme dadurch tatsächlich etwas lesbarer. Den gleichen Effekt erreicht man aber auch durch eine disziplinierte Programmierung mit anderen Sprachen. Beim strukturellen Aufbau eines Simulationsprogramms, der Implementierung der Ablaufsteuerung sowie der gesamten Ein-/Ausgabe erfährt der Programmierer jedoch nach wie vor keine direkte Unterstützung. Eine Annäherung an die Modellwelt des Anwenders leisten diese Sprachen überhaupt nicht. Der komponentenweise Aufbau kontinuierlicher Modelle ist zudem gänzlich unmöglich, da man hierfür andere Kommunikationsmechanismen benötigt.

## Simulationspakete

Simulationspakete sind Programmpakete für Simulationsanwendungen. Sie sind in einer höheren Programmiersprache geschrieben und unterstützen den Anwender beim Aufbau und der Strukturierung von Simulationsprogrammen.

Dazu stellen sie dem Programmierer einen Programmrahmen zur Verfügung, in den er nur noch die Anweisungen eintragen muß, welche die Dynamik seines Modells repräsentieren.

Alle übrigen Dienste werden von einer Sammlung von Unterprogrammen übernommen. Insbesondere wird eine Ablaufsteuerung und eine Verwaltung für Zeitereignisse bereitgestellt. Weiterhin werden diverse numerische Verfahren, vor allem zur Lösung von Differentialgleichungen oder zur Erzeugung von Zufallszahlen und evtl. auch Verfahren zur Auswertung und Darstellung der Ergebnisse angeboten.

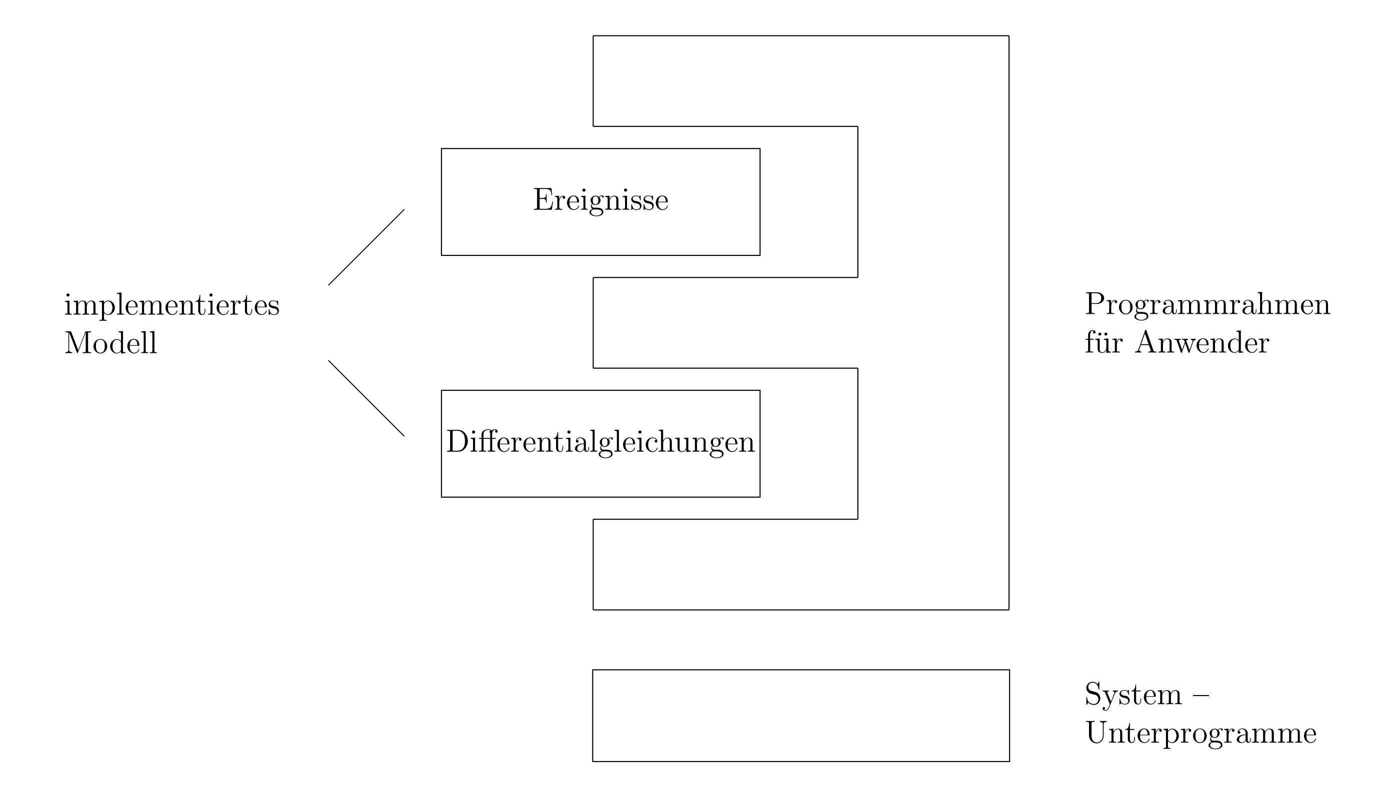



Abb. 1.3-1: Prinzipieller Aufbau eines Simulationspakets

Die Programmierung diskreter Modelle wird ggf. durch die Vorgabe einer Modellwelt unterstützt, für die ebenfalls bereits eine Implementierung in Form von Unterprogrammen vorliegt. So bietet das Simulationspaket GPSS-FORTRAN [Schm 84a,b] die Modellwelt der Simulationssprache GPSS [Schr 91], bestehend aus Transactionen und diversen Stationstypen wie Facility, Pool oder Storage.

Weitere Vertreter für Simulationspakete sind GASP IV [Prit 74] und BORIS [BORIS 85].

Die vorgegebene Systematik entbindet den Programmierer davon, eine eigene Programmstruktur entwerfen und häufig wiederkehrende Programmteile selbst entwickeln zu müssen.

Simulationspakete bemühen sich darum, durch einen geschickten Aufbau die Implementierung des Modells von der Ablaufsteuerung und den übrigen Systemprogrammen zu trennen. Dadurch wird eine Modellimplementierung leichter nachvollziehbar. Für die Programmierung des Modells und auch für das Verständnis des Modellablaufs ist die Kenntnis der Ablaufsteuerung allerdings notwendig. Simulationspakete bilden auch die Grundlage für datengesteuerte Simulatoren und Simulationssprachen.

## Datengesteuerte Baustein-Simulatoren

Beschränkt man sich auf ein sehr enges Anwendungsgebiet, kann man in vielen Fällen Simulationsprogramme so gestalten, daß die Konfiguration des Modells in Form von (tabellarischen) Daten beschrieben werden kann. Ohne daß eine erneute Übersetzung erforderlich wäre, kann ein Modell erstellt und verändert werden. Diese Vorgehensweise setzt voraus, daß das Modell aus genau spezifizierten Modellbausteinen aufgebaut werden kann, die miteinander über vordefinierte Kanäle kommunizieren.

Als Beispiele für derartige Modellwelten lassen sich anführen:

- Warteschlangennetze bestehend aus Quellen, Senken, Bedieneinheiten mit Warteschlangen und Verzweigungen.
- Petrinetze bestehend aus Stellen und Transitionen
- Neuronale Netze
- Elektrische Schaltkreise bestehend aus Quellen, Widerständen, Kapazitäten, Induktivitäten etc.

Zusätzlich können für jeden Modellbaustein eine Vielzahl von Parametern angegeben werden. Auch unterschiedliche Strategien sind über Parameter anwählbar.

Im Sinne der Graphentheorie lassen sich die Bausteinen als Knoten und die Kommunikationskanäle als Kanten ansehen. Hieraus wird unmittelbar einsichtig, daß sich derartige Modellwelten stets auch graphisch repräsentieren lassen. In Verbindung mit komfortablen graphischen Bedienoberflächen kann jeder Anwender bei Kenntnis der Modellwelt zügig Modelle aufbauen und in ihrer Struktur verändern. Das graphisch aufgebaute Modell-Layout läßt sich gleichzeitig für die Animation heranziehen.

Seitdem nahezu jeder Anwender einen graphischen Rechner-Arbeitsplatz besitzt, sind eine Vielzahl derartiger Simulatoren entstanden. Diese enthalten neben elementaren Bausteinen wie sie in den Beispielen oben angeführt wurden auch komplexere Komponenten, um den Bedürfnissen der Anwender Rechnung zu tragen. Auf dem Gebiet der Fertigungstechnik sind die derzeit wichtigsten Vertreter DOSIMIS III, WITNESS und SIMPLE++. Bei kontinuierlichen Anwendungen steht XANALOG für den Stand der Technik.

Wegen ihrer komfortablen Bedienung haben datengesteuerte Simulatoren Maßstäbe gesetzt. Alle Simulatoren werden mittlerweile an ihnen gemessen, weil der Anwender selbst damit umgehen kann. Ob er auch alle seine Probleme damit lösen kann, bleibt meist dahingestellt.

Durch die Beschränkung auf bestimmte Modellbausteine, lassen sich nämlich viele Eigenarten eines Systems nicht oder nur sehr trickreich nachbilden. Auf ein bestimmtes eng umrissenes Anwendungsgebiet ist man ohnehin verwiesen. Daher bedeutet ein Wechsel des Anwendungsgebietes für den Benutzer, daß er einen anderen (datengesteuerten) Simulator verwenden muß. Ein weiterer Nachteil liegt darin, daß Steuerungs-Strategien nicht oder nur bei spezieller Systemkenntnis in das Modell eingebracht werden können.

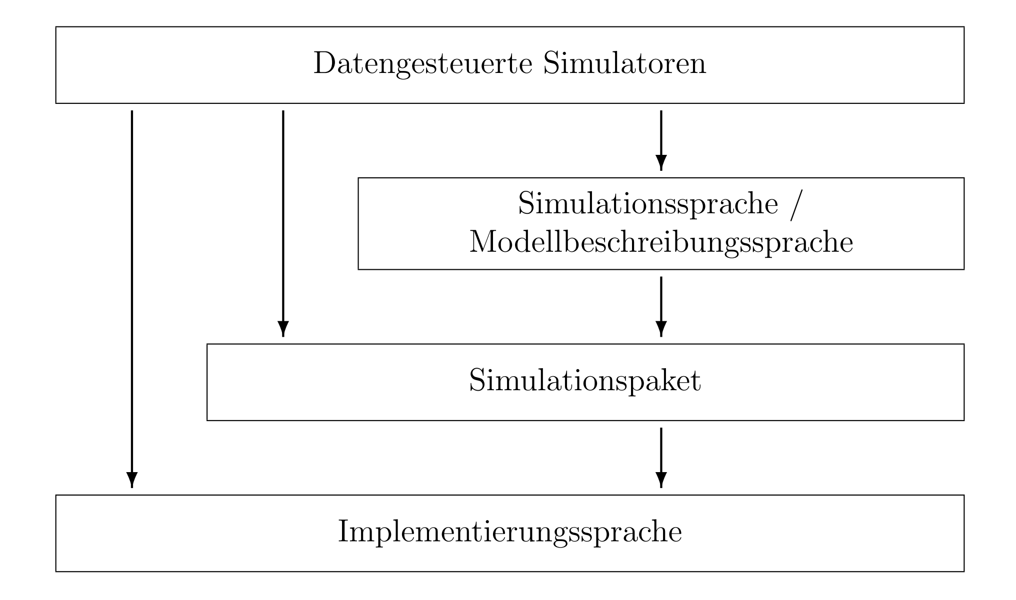



Abb. 1.3-2: Werkzeuge zur Erstellung von Simulationsprogrammen

## Simulationssprachen

Schon mit Beginn der sechziger Jahre entstanden die ersten Simulationssprachen: GPSS [Schr 91], SIMSCRIPT [Russ 83], SLAM [PriPed 79] und SIMAN [Ped86] auf dem Gebiet der diskreten Simulation (Vergleich in [Prit 82]) und CSSL, CSMP und ACSL auf Gebiet der kontinuierlichen Simulation.

Alle Sprachen zur Formulierung diskreter Modelle ermöglichen eine prozeß-orientierte Beschreibung. Man beschreibt gewissermaßen die Lebensgeschichte eines Auftrags (Werkstücks, Fahrzeugs etc.), der in Warteschlangen (Puffer, Abstellplätze etc.) ruht und von Betriebsmitteln (Maschinen, Streckenabschnitten) zeitverzögert bearbeitet wird (transactions-orientierte Simulation).

Mit Ausnahme von SIMSCRIPT lassen sich die Betriebsmittel nicht individuell modellieren, sondern nur aus vorgefertigten Beschreibungsblöcken zusammensetzen. Durch diese Vereinfachung wird allerdings eine komfortable Handhabung der Modelle auch erst möglich.

Da der prozeß-orientierte Ansatz durch diese Einschränkungen nicht mehr alle Anwendungsfälle abdeckt, bieten SLAM und SIMAN (beide beruhen auf dem Simulationspaket GASP) daneben eine ereignis-orientierte Form der Modellbeschreibung an. Die gleiche Erweiterung wurde in GPSS-FORTRAN durchgeführt. Diese Vorgehensweise führt aber beim kombinierten Einsatz beider Konzepte zu methodologischen Ungereimtheiten.

Diese Sprachen sowie ihre Weiterentwicklungen umfassen nicht nur eine Beschreibung des Simulationsmodells, sondern auch Befehle zur Durchführung der Experimente und zur Ausgabe der Ergebnisse. Neue Experimente machen zumindest eine teilweise Neu-Übersetzung erforderlich.

Erst zu Beginn der achtziger Jahre wurde massiert auf diesen Mißstand hingewiesen und die Forderung nach einer Trennung von Modell und Experiment erhoben.

Andere Schwierigkeiten ergaben sich wegen des geringen Sprachumfangs und insbesondere wegen der begrenzten Möglichkeiten, Ein-/Ausgabe zu betreiben. Aus diesem Grund ge-

statten manche Simulationssprachen Eingriffe in das darunterliegende Programmpaket, eine Vorgehensweise die sicherlich nicht zur Übersichtlichkeit beiträgt.

Neuere Entwicklungen auf dem Gebiet der Simulationssprachen, die auch einen modularen Modellaufbau gestatten sollten, erfolgten jedoch nur sehr schleppend und erlangten keine Produktreife (COSMOS [Kett83, Kett88], SYSMOD [SYSMOD 86]). Die ganze Aufmerksamkeit galt in den vergangenen zehn Jahren den datengesteuerten Simulatoren mit ihren beeindruckenden graphischen Oberflächen.

Die Abbildung 1.3-2 zeigt, in welcher Weise Werkzeuge zur Erstellung von Simulationsprogrammen aufeinander aufbauen.

Die Ausführungen machen deutlich, daß keiner der bisherigen Ansätze eine allgemein zufriedenstellende Lösung gebracht hat. Wie wir später sehen werden, können vor allem neuere Anforderungen damit nicht bearbeitet werden.

## 1.4 Ansatz für ein neues Sprachkonzept

Es wurde bereits herausgestellt, daß alle Bestrebungen darauf hinauslaufen, die Kluft zwischen einer mehr oder weniger vagen Modellvorstellung des Anwenders und dem mehr oder weniger kryptischen Simulationsprogramm eines Programmierers zu überbrücken.

Die Entwickler der Simulationssprachen bemühten sich, anstelle der höheren Programmiersprachen eine (prozedurale) Sprache anzubieten, welche durch geeignete Sprachkonstrukte die Formulierung von prozeß-orientierten Simulationsprogrammen vereinfacht. Wie bereits oben ausgeführt, erscheint der Erfolg fraglich, weil nicht alle Modellvorstellungen auf die angebotene, einfache Weise beschreibbar sind. Um weitere Eingriffsmöglichkeiten ausnutzen zu können, sind jedoch wiederum beträchtliche Systemkenntnisse vonnöten.

Der in dieser Arbeit anvisierte Sprachentwurf geht nicht von einer prozeß-orientierten, sondern von einer ereignis-orientierten Beschreibung dynamischer Modelle aus. Wir schließen uns damit einer zustands-orientierten Betrachtung an, wie sie bei der Beschreibung kontinuierlicher Modelle seit langem üblich ist. Dies geschieht in der Erwartung, auf diese Weise eine Sprache definieren zu können, die elementar genug ist, um alle Arten von dynamischen Vorgängen beschreiben zu können.

Auf diese Weise wird zwischen Modellvorstellung und prozeduralem Simulationsprogramm eine weitere Sprachschicht eingezogen, die das dynamische Modell vollständig formal beschreibt.

Die – nicht formale und mehr oder weniger mathematisierte – Modellvorstellung wird dadurch konkretisiert und formalisiert. Aus dieser Modellbeschreibung läßt sich schließlich das Simulationsprogramm durch einen Compiler automatisch generieren.

Um dem Begriff Modellbeschreibungssprache gerecht zu werden, müssen vor allem vier Zielsetzungen erreicht werden:

- Die formale Beschreibung eines Modells muß der gedanklichen Vorstellung bzw. einer bereits vorliegenden mathematischen Beschreibung möglichst nahe kommen. Die Notation muß gut lesbar sein. Dazu muß sie ausführlich, darf aber nicht ausschweifend sein.

- Die Reihenfolge von Abläufen muß man der Modellbeschreibung selbst entnehmen können. Kenntnisse über die Arbeitsweise der Ablaufsteuerung dürfen nicht vonnöten sein.
- Die Modellbeschreibung darf keine Information enthalten, die für die Ein/Ausgabe und die Bedienung des Simulationsprogramms dienlich ist. Diese Aufgaben sind durch datengesteuerte Standardschnittstellen abzudecken. Eine Ausnahme bilden lediglich solche Ein-/Ausgaben, die vom Simulationsmodell selbst initiiert werden.
- Durch semantische Prüfungen muß sichergestellt werden können, daß die Formulierungen, die ein Modell beschreiben, vollständig und widerspruchsfrei sind.

Im Sinne der Informatik handelt es sich also um eine Spezifikation des Simulationsprogramms. Eine Modellbeschreibungssprache ist daher eine Spezifikationssprache für Simulationsmodelle. Wir unterscheiden deshalb fortan zwischen der Modellimplementation (Simulationsprogramm) und der Modellbeschreibung (Modellspezifikation).

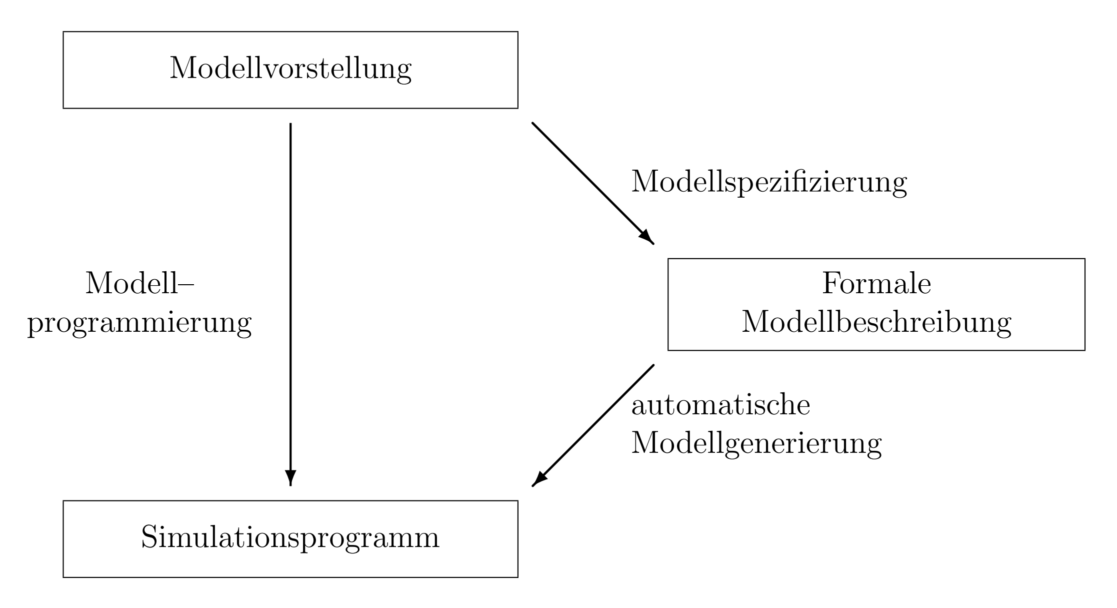



Abb. 1.4-1: Modellerstellung durch Spezifizierung

Damit eine Modellbeschreibungssprache auch im praktischen Einsatz erfolgreich sein kann, muß sie drei weitere Anforderungen erfüllen:

- Die Sprache muß universell genug sein, um eine breite Anwendungspalette abzudecken. Der Anwender möchte die Gewißheit haben, daß sein Problem tatsächlich mit Hilfe der Sprache behandelt werden kann und er nicht irgendwann auf unüberwindliche Schwierigkeiten stößt. Und er möchte die Sprache nicht nur für einen Modelltyp einsetzen, sondern auch für andere Anwendungen. Nur dann lohnt sich für ihn der Aufwand, die Sprache zu erlernen.
- Modelle müssen hierarchisch modularisierbar sein. Dies ermöglicht einerseits, Modelle aus vorgefertigten Bausteinen zusammenzusetzen, und andererseits, Modelle in überschaubare Komponenten zu zerlegen. Jede Komponente muß ein eigenständiges Modell repräsentieren, um für sich getestet werden zu können. Dies sichert eine hohe Zuverlässigkeit der zusammengesetzten Modelle.

- Die Ausführung eines automatisch generierten Simulationsprogramms muß effizient erfolgen. Ein eventueller Verlust an Rechengeschwindigkeit gegenüber einem "von Hand" erstellten Simulationsprogramm muß tolerierbar sein.

Weitere, für die Praxis wichtige Erfordernisse, die aber nicht den Sprachentwurf beeinflussen, sind in Kapitel 6 zusammengestellt.

Im Bereich kontinuierlicher Modelle kann sich eine formale Modellbeschreibung eng an die üblichen Darstellungsmethoden in Form von Differentialgleichungen und algebraischen Gleichungen anlehnen. Die Systemtheorie [Pich 75, Unbe 80, Wun 86] hat aufgezeigt, wie sich solche Modelle auch modular zerlegt repräsentieren lassen. In [Ören 84] wird mit GEST ein erster Sprachentwurf dokumentiert, der diese Überlegungen aufgreift. Mit SIMPLEX-MDL [Esch 90] entstand schließlich die erste vollständig implementierte Sprache zur Formulierung und Simulation diskret/kontinuierlicher Modelle.

Im Bereich der Warteschlangen- und Transportmodelle fehlen derartige Vorbilder. Zwar gibt es eine Reihe von Ansätzen, auch graphischer Art, an einer allgemeinen Akzeptanz mangelt es jedoch bislang.

Darstellungen aus theoretischen Abhandlungen beschränken sich auf Modelle, die sich analytisch behandeln lassen, und sind daher nicht allgemein genug. Darstellungen aus Simulationssprachen (GPSS, SLAM, SIMAN) sind – den Sprachkonzepten entsprechend – prozeßorientiert und aus diesem Grund nicht brauchbar.

Diese Abhandlung will einen Weg aufzeigen, den angeführten Anforderungen gerecht zu werden. Sie stellt den Versuch dar, das im Bereich der kontinuierlichen Modelle so erfolgreiche Konzept der zustandsorientierten Modellbeschreibung auch auf die Welt der diskreten Warteschlangen- und Transportmodelle zu übertragen.

Dieses Bemühen, beide Welten zu vereinheitlichen, ist aus verschiedenen Gründen reizvoll:

In vielen Anwendungsgebieten entstehen auf unterschiedlichen Abstraktionsstufen diskrete bzw. kontinuierliche Modelle. Mikroskopische Modelle sind oft diskreter Natur, wenn die Wechselwirkungen einzelner Individuen betrachtet werden. Abstrahiert man von diesem individuellen Verhalten, erhält man meist kontinuierliche makroskopische Modelle (z.B. kinetische Gastheorie). Umgekehrt kann auch das mikroskopische Verhalten kontinuierlicher und das makroskopische Verhalten diskreter Natur sein (z.B. Digitalrechner).

Eine einheitliche Beschreibungsform erleichert sicherlich das Umdenken zwischen diesen Modellformen. Aber auch eine Kombination diskreter und kontinuierlicher Modelle wird dadurch denkbar.

Schließlich sind aber auch die methodologischen Gesichtspunkte interessant. Welche Gemeinsamkeiten zeichnet das Denken in kontinuierlichen und diskreten Modellen aus und wo liegen die Unterschiede ? Können neuartige Beschreibungsformen für diskrete Modelle die traditionellen Beschreibungsformen für kontinuierliche Modelle befruchten ? Wie sehr ähneln sich Syntax und Semantik zur Beschreibung kontinuierlicher bzw. diskreter Vorgänge ?

Die angestrebte Spezifikationssprache soll eine manuelle Programmierung des Simulationsmodells vollständig ersetzen. Die Kluft zwischen Modellvorstellung und Modellbeschreibung muß deutlich kleiner sein als zwischen Modellvorstellung und Simulationsprogramm. Sie soll klein genug sein, damit auch ein in formalen Sprachen wenig erfahrener Anwender damit umgehen kann.

## 1.5 Neue Anforderungen an die Simulationstechnik am Beispiel der Fertigungstechnik

Die Anregungen für neue Werkzeuge zum Erstellen von Simulationsprogrammen kamen in den letzten Jahren hauptsächlich aus der Fertigungstechnik, die neuartigen Anforderungen [INGER 85] gegenübersteht und sich hierfür Lösungen durch Simulationsstudien erhofft.

Die Anforderungen der Fertigungstechnik sind besonders anspruchsvoll und schließen die Bedürfnisse der meisten anderen Anwendungsgebiete der diskreten Simulation mit ein.

Damit sich der Leser eine Vorstellung davon machen kann, in welcher Weise man in Zukunft Simulationstechnik zum Einsatz bringen möchte und welche Leistungsfähigkeit gefordert wird, sollen die gestellten Anforderungen im Rahmen dieser Einleitung abschließend kurz umrissen werden.

Die Anforderungen der Anwender lassen sich in Anforderungen an

- die Funktionalität und an
- die Ergonomie

unterscheiden.

Diese beiden Anforderungskreise sind naturgemäß widersprüchlich: Je benutzerfreundlicher ein Programm ist, desto schwieriger ist es, diesem auch einen universellen Funktionsumfang mitzugeben. Andererseits ist es ebenso schwierig, universelle Funktionalität auch dem ungeübten Anwender zur Verfügung zu stellen.

Die neuen Anforderungen an die Simulationstechnik [Schm 88] resultieren aus der Notwendigkeit, neuartige Produktionsanlagen modellieren zu müssen, und aus dem Wunsch, durch eine komfortablere Software-Umgebung auch dem fertigungstechnischen Planer die Simulation zur Verfügung zu stellen.

Die Zielsetzung für einen Einsatz der Simulation hingegen ist nach wie vor dieselbe: Es soll untersucht werden, auf welche Weise die geforderte Leistungsfähigkeit einer Anlage bei geringsten Kosten erreicht werden kann. Zu diesem Zweck werden alternative Konstruktionen untersucht, die Dimensionierung der einzelnen Anlagenteile aufeinander abgestimmt und nach geeigneten Betriebsabläufen und Strategien gesucht.

### 1.5.1 Anforderungen an die Funktionalität

Planer von fertigungstechnischen Anlagen streben von jeher eine Lösung an, welche die vorgegebene Produktionsrate bei möglichst geringen Kosten (Investitions- und Betriebskosten) erreicht.

Bei einer großen Stückzahl von Teilen, geringer Variantenvielfalt und einem großen Fertigteilelager kann ein sehr kontinuierlicher Fertigungsprozeß erreicht werden. Störungen im Fertigungsablauf lassen sich durch genügend große Puffer zufriedenstellend überbrücken. Die Produktionsrate als einzige interessierende Größe läßt sich sehr leicht aus den Verfügbarkeiten der Maschinen ermitteln.

Durch die immer individueller werdenden Kundenwünsche und die daraus resultierende Variantenvielfalt hat sich die Situation in den letzten Jahren zunehmend verändert. Es ist nicht mehr möglich, jedes gewünschte Produkt auf Lager zu halten, da das Lager zu groß und zu

teuer, die Kapitalbindung beträchtlich und das Risiko, bereits produzierte Ware gar nicht mehr absetzen zu können, zu hoch wäre.

An die Stelle einer Produktion auf Vorrat tritt daher zunehmend eine Produktion auf Anfrage. Diese bei kleinen Stückzahlen schon immer praktizierte Produktionsform findet demnach verstärkt Eingang im Bereich der Serien- und Massenfertigung.

Bei kleinen Stückzahlen kann man es sich leisten, auf eine detaillierte Planung des Fertigungsprozesses zu verzichten, weil der Automatisierungsgrad gering ist und man sich die Intelligenz der Werker zunutze machen kann, die das Geschehen noch überblicken können. Bei großen Stückzahlen lohnt sich die Anschaffung von weitgehend bedienerlosen Fertigungsautomaten, Transporteinrichtungen und Läger. Ihr Einsatz muß im Voraus wohl durchdacht sein, um Fehlinvestitionen zu vermeiden.

Da auf Anfrage produziert wird, interessiert nicht mehr nur der erreichbare Durchsatz (Produktionsrate), sondern vor allem auch die Durchlaufzeiten der Teile, da man den Kunden nicht lange auf seine Bestellung warten lassen will.

Daneben ist die notwendige Anzahl an Pufferplätzen zu ermitteln, um einen reibungslosen Betrieb und damit eine hohe Auslastung der Maschinen zu gewährleisten. Auch die erforderliche Leistung der Transporteinrichtungen ist interessant, damit der Betriebsablauf nicht verzögert wird.

Solange jede Produktionseinrichtung nur einen einzigen Bearbeitungsgang übernimmt, können diese durch starre Transporteinrichtungen aneinandergereiht werden. Übernimmt jedoch eine Maschine mehrere Bearbeitungsgänge oder ist die Maschinenfolge nicht für alle Varianten dieselbe, müssen die Maschinen flexibel miteinander verkettet werden.

In diesem Fall kann es passieren, daß ein Teil vorübergehend nicht weiterverarbeitet werden kann, weil die dazu nötige Maschine noch belegt ist. Umgekehrt kann es vorkommen, daß eine Maschine warten muß, weil mit dem Zuarbeiten eines Teils nicht rechtzeitig begonnen wurde. Im Gegensatz dazu ist bei starr verketteten und getakteten Anlagen ein zügiges Durchschleusen stets gewährleistet. Es kann auch zu Verklemmungen kommen, wenn beispielsweise zwei Teile auf der jeweils anderen Maschine weiterzubearbeiten sind und kein Ablageplatz mehr frei ist.

Um einen derart komplizierten Fertigungsablauf bewältigen zu können, wird mittlerweile in der Produktion eine größere Anzahl an Rechnern eingesetzt, welche die diversen Dispositions- und Steuerungsaufgaben übernehmen. Je höher die Flexibilität der Anlage ist, desto größer ist die Bedeutung der Leit- und Steuerungssysteme. Die von ihnen eingesetzten Strategien müssen möglicherweise detailgetreu nachgebildet werden, wenn man spezielle Effekte untersuchen möchte.

Simulationsstudien sind ein geeignetes Mittel, die Planung solch komplexer und schwer durchschaubarer Anlagen zu erleichtern und Fehlinvestitionen zu vermeiden. Wegen der hohen Vielfalt an Einrichtungen zum Fertigen, Montieren, Transportieren und Lagern und der besonderen Bedeutung der Steuerungsalgorithmen, die in jeder Anlage individuell sind, ist jedoch eine so breite Funktionalität erforderlich, wie es eigentlich nur von einem universell einsetzbaren Werkzeug erwartet werden kann. Spezielle Simulatoren für die Fertigungstechnik können stets nur für sehr spezielle Fragestellungen zum Einsatz kommen.

Die Gegebenheiten auftrags-orientierter Fertigungsanlagen entsprechen ziemlich genau denen, wie sie von Betriebssystemen für Rechenanlagen her bekannt sind. Hält man sich die dort

vertretene Modellwelt vor Augen, die aus Aufträgen sowie aktiven und passiven Betriebsmitteln besteht, bekommt man eine konkretere Vorstellung von der erforderlichen Funktionalität.

Aktive Betriebsmittel (Stationen) nehmen Aufträge entgegen und führen diese - evtl. unter Verwendung weiterer Betriebsmittel - aus. Ein an einer Station ankommender Auftrag wartet zunächst, bis er bedient werden kann. Die Station bearbeitet dann diesen Auftrag gemäß dessen Arbeitsplan Schritt für Schritt mit dafür vorgesehenen Zeitverzögerungen. Der Arbeitsplan ist ein zyklenfreier, gerichteter Graph, dessen Knoten die Bearbeitungsschritte und dessen Kanten die Abhängigkeiten von vorangegangenen Bearbeitungsschritten darstellen. Für Graphen dieser Art findet man viele Bezeichnungen, beispielsweise Vorranggraph, Abhängigkeitsgraph oder Reihenfolgegraph.

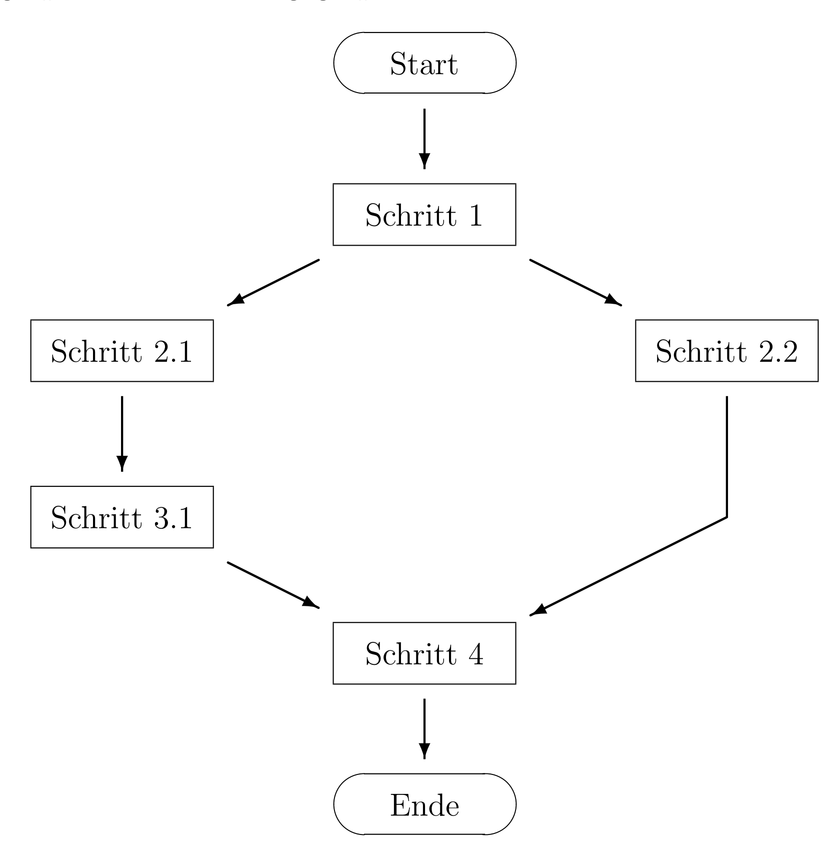



Abb. 1.5-1: Arbeitsplan als Vorranggraph

Nachdem mit der Bearbeitung begonnen wurde, kann der Auftrag bzw. der Bearbeitungsschritt entweder

- vollständig zu Ende geführt werden,
- unterbrochen werden, weil ein wichtigerer Auftrag zu bearbeiten ist (Verdrängung),
- unterbrochen werden, weil eine vorgegebene Zeitscheibe abgelaufen ist,
- blockiert werden, weil auf die Zuteilung eines Betriebsmittels gewartet werden muß oder
- blockiert werden, weil auf die Bearbeitung eines Unterauftrags gewartet werden muß.

Wartende und blockierte Aufträge werden in Warteschlangen zurückgestellt.

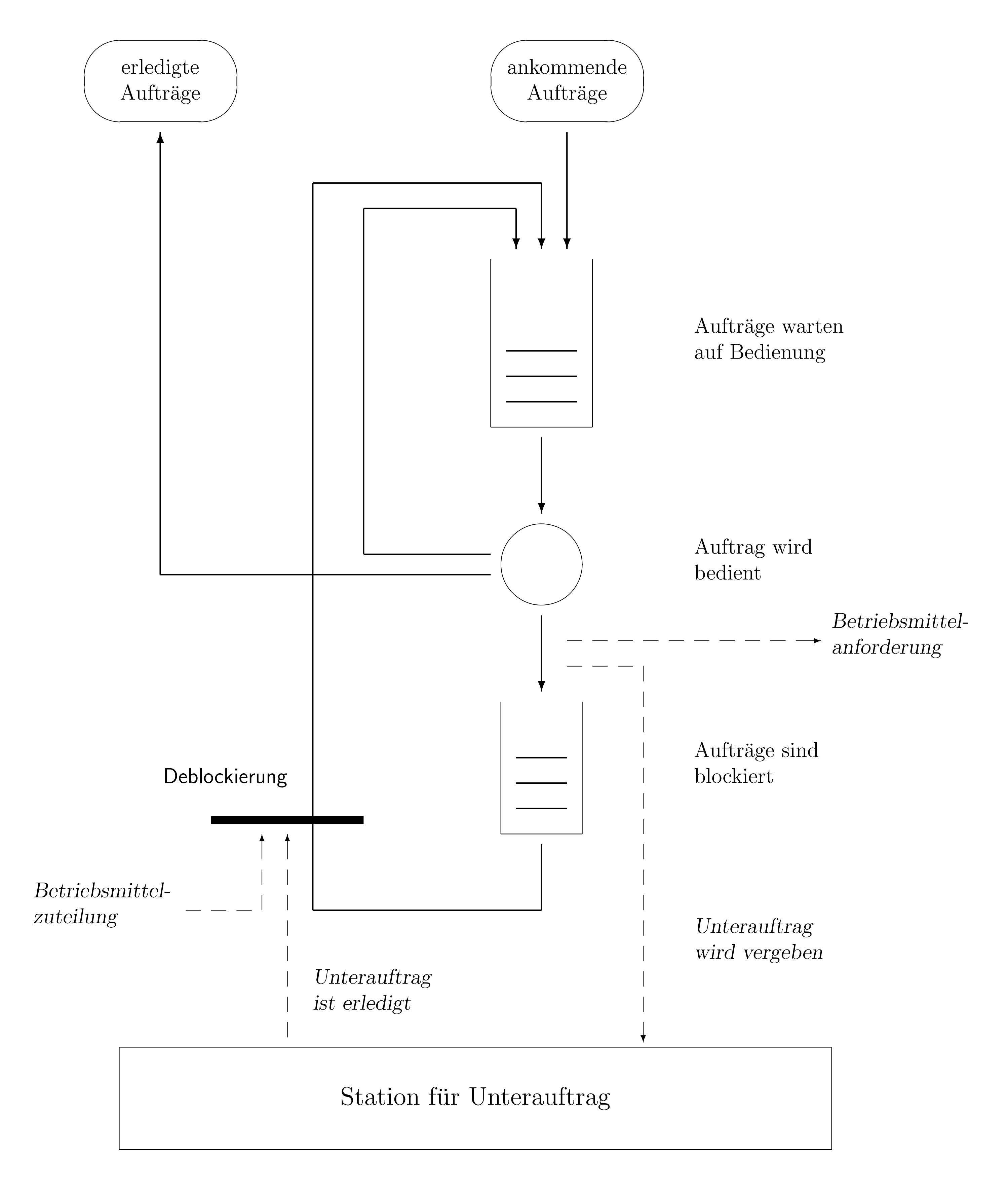



Abb. 1.5-2: Funktionsweise einer Station zur Bearbeitung von Aufträgen

Sind für einen Arbeitsschritt Betriebsmittel erforderlich, so werden diese angefordert. Wenn alle benötigten Betriebsmittel zugeteilt sind, kann die Station den Auftrag bearbeiten. Solange die Station noch auf Betriebsmittel warten muß, ist der Auftrag blockiert. Ob damit auch die Station blockiert ist oder in der Zwischenzeit ein anderer, unabhängiger Bearbeitungsschritt ausgeführt wird, hängt von der individuellen Anwendung ab.

Ein Betriebsmittel bzw. dessen Verwaltung nimmt Anforderungen der Stationen entgegen. Sobald es verfügbar ist, wird es einer Station zugeteilt. Wenn es die Station nicht mehr benötigt, gibt sie es wieder frei.

Eine Station muß einen Bearbeitungsschritt nicht selbst ausführen, sie kann ihn auch an eine andere Station delegieren. Zu diesem Zweck erzeugt sie einen Unterauftrag, wählt eine mögliche Station aus und schickt ihr den Unterauftrag zur weiteren Bearbeitung. Wenn nicht in der Zwischenzeit ein anderer Bearbeitungsschritt erledigt werden kann, ist der Auftrag solange blockiert. Sobald der erledigte Unterauftrag zurückgeliefert wird, kann der ursprüngliche Auftrag wieder deblockiert werden.

Auf den Fortschritt der Bearbeitung eines Auftrag hat es demnach die gleiche Wirkung, ob Betriebsmittel angefordert werden oder ein Unterauftrag erzeugt wird.

Neben dieser Auftragsabwicklung ist auch die Betriebsmittelnutzung zu modellieren. Ein Betriebsmittel steht nicht die gesamte Zeit für die Bearbeitung eines Auftrags zur Verfügung, weil es möglicherweise auf- oder abgerüstet wird oder eine Störung hat.

Auch die Betriebsmittelzuteilung ist von Bedeutung, da eine Zuteilung möglicherweise an Bedingungen geknüpft ist. Die Zuteilung eines Fahrzeugs ist beispielsweise nur dann möglich, wenn es eine bestimmte Größe besitzt.

Die Modellierung der Auftragsabwicklung und die Modellierung der Betriebsmittelnutzung liefern uns gerade die Leistungsgrößen eines Systems. Diese unterscheiden wir in primäre Leistungsgrößen, welche für den Auftraggeber interessant sind, und in sekundäre Leistungsgrößen, an denen der Betreiber der Anlage Interesse hat. Die primären Leistungsgrößen wie Wartezeiten, Bedienzeiten und Blockierzeiten erhält man aus der Auftragsabwicklung, die sekundären Leistungsgrößen wie Auslastung, Durchsatz und Lagerbestände kann man der Betriebsmittelnutzung entnehmen.

Aus diesen Betrachtungen läßt sich auf die notwendige Funktionalität einer Modellbeschreibungssprache schließen.
Erforderliche Datentypen:

- für Zustände: Variablen mit unterschiedlichen Wertemengen
- für Aufträge: Verbunde sowie Felder oder Listen von Verbunden
- für Warteschlangen: Listen von Verbunden

Erforderliche Funktionen:

- Zustandswechsel:
  - deterministisch (zustandsbedingt und zeitbedingt)
  - stochastisch (Bestimmung von Folgezustand und Zustandsdauer)
- Entnehmen von Aufträgen aus Warteschlangen nach beliebigen Strategien
- Einreihen von Aufträgen in Warteschlangen nach beliebigen Kriterien
- Erzeugung und Vernichtung von (Unter-) Aufträgen
- Senden und Empfangen von (Unter-) Aufträgen

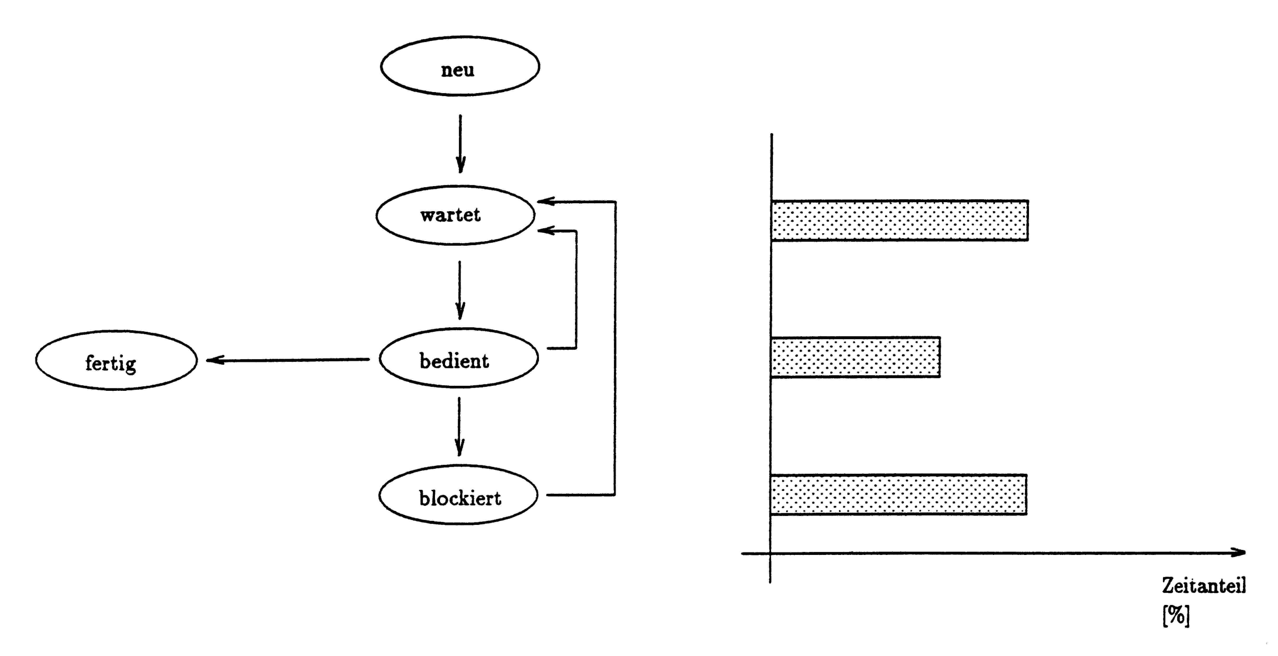



Abb. 1.5-3: Primäre Leistungsgrößen aus Zuständen der Aufträge

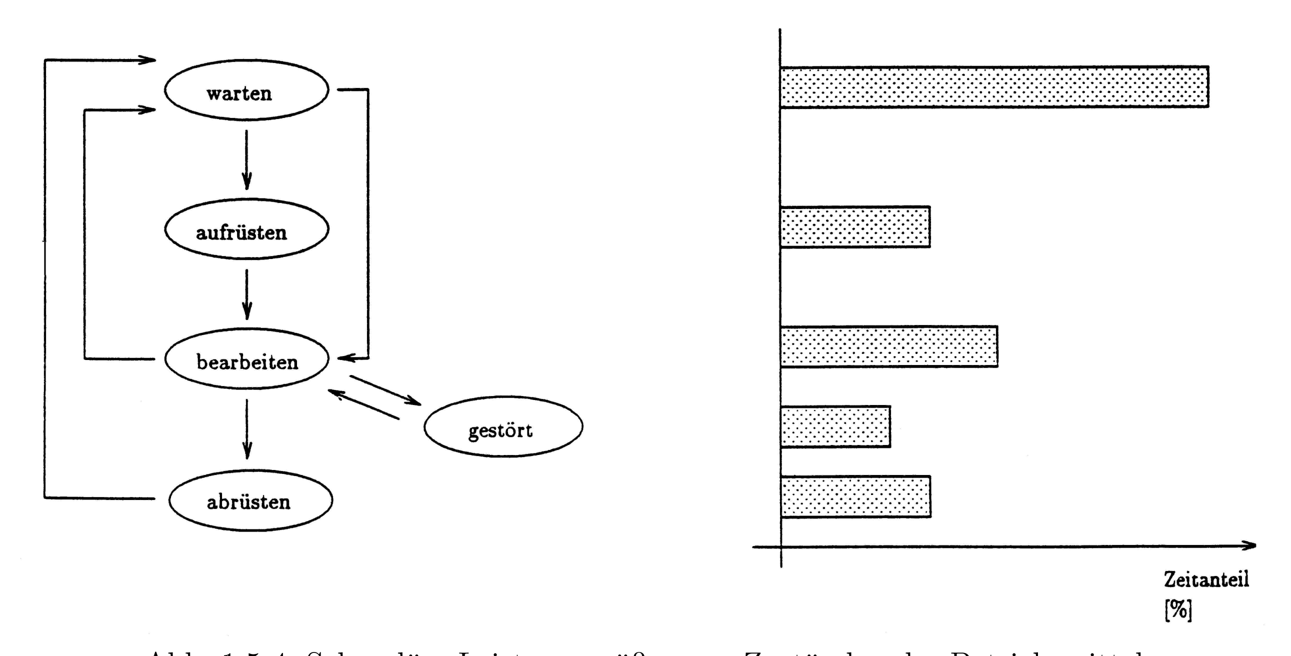



Abb. 1.5-4: Sekundäre Leistungsgrößen aus Zuständen der Betriebsmittel

– Delegieren von Aufträgen an Stationen nach beliebigen Auswahlstrategien

Von genau den gleichen Betrachtungen ist übrigens die Simulationssprache GPSS motiviert. Allerdings ist es dort nur möglich, die Auftragsabwicklung zu modellieren. Die Betriebsmittelzuteilung und Betriebsmittelnutzung kann nicht individuell beschrieben werden. Die Mächtigkeit der Sprache ist deshalb in keinster Weise ausreichend, um die notwendige Funktionalität in allgemeiner Weise herstellen zu können.

### 1.5.2 Anforderungen an die Ergonomie

Zu Beginn der achtiger Jahre entstand das CIM-Konzept (computer integrated manufacturing), das den Zeitraum von der Idee eines Produkts bis zu dessen Fertigung auf automatisierten Produktionseinrichtungen deutlich verkürzen sollte.

Zu diesem Zweck sollten die Entwickler eines neuen Produkts, die Planer der Fertigungsanlagen sowie die Betreiber der Fertigungsstätten auf eine gemeinsame Datenbasis zurückgreifen können.

Dort gespeicherte Daten sollten auch die Grundlage für eine Simulation der geplanten Produktionsanlagen sein. Der Anwender sollte für die Simulation nur noch ein Minimum an Daten ergänzen müssen, um seine Ergebnisse als Animation oder Präsentation auf dem Bildschirm zu sehen.

Neben der Modellerstellung aus einer gemeinsamen Datenbasis wurden noch beträchtliche andere Anforderungen genannt. Entsprechend dem physikalischen Aufbau einer Fertigungsanlage sollte das Gesamtmodell aus seinen Systemkomponenten zusammengesetzt werden können.

Bei dieser Vorgehensweise sollte selbstverständlich die Denkwelt des Planers nicht verlassen werden. Er sollte seine Komponenten als bekannte graphische Symbole wiederfinden, um mit diesen intuitiv Modelle aufzubauen und zu experimentieren. Arbeitspläne, Entscheidungstabellen und was sonst alles nötig ist, sollte in gewohnter Form eingegeben werden können.

Hinter diesen konkreten Wünschen nach Anschaulichkeit stehen letztendlich die verständlichen Forderungen nach

– kurzer Einarbeitungszeit
– zügiger und sicherer Bedienung
– schnellem Reaktivieren des abgelegten Wissens
– rechtzeitiger und verständlicher Information über fehlerhafte Bedienung.

Da abzusehen war, daß die Modelle womöglich sehr umfangreich werden können, war von Anfang an auch auf eine effiziente Ausführung der Simulationsläufe zu achten.

Auf Seiten der Betriebsleitung entstand die Wunschvorstellung, eine gesamte Fabrik zu simulieren, um sich mehr Transparenz über die Abläufe in ihrer Firma verschaffen zu können. Anders als in der Vergangenheit wirken sich Störungen im geplanten Ablauf bei modernen Konzepten wie *just in time* viel schneller und intensiver auf die nachfolgenden Abteilungen aus.

Auch wenn das Ziel der Fabriksimulation etwas utopisch erscheint, so hat es doch insofern seine Berechtigung, daß es möglich sein sollte, die Teilmodelle eines Betriebs miteinander zu verbinden.

Hierzu sollten alle Modelle in einer gemeinsamen Umgebung laufen und auf der gleichen Basis beruhen. Teilmodelle sollten auf verschiedenen Abstraktionsstufen modellierbar und gegeneinander austauschbar sein. Auf diese Art lassen sich sowohl Detailuntersuchungen als auch globale Betrachtungen durchführen.

### 1.5.3 Diskussion der Lösungsansätze

Aus dem bisher gesagten wird klar, daß jede Art von Programmiersprache, auch eine objektorientierte, und auch ein Simulationspaket nicht geeignet ist, diese Ansprüche zu erfüllen. Programmiersprachliche Konzepte wie Zeiger sind für einen Nicht-Informatiker nicht beherrschbar! Dazu fehlt ihm das Grundwissen und vor allem auch die nötige Übung.

Seitdem die genannten Anforderungen im Rahmen der Überlegungen zu CIM-Konzepten aufgestellt und propagiert wurden, gab es vier größere Projekte, mit denen man versuchte, zu einsatzfähigen Lösungen zu kommen.

DOSIMIS III, entwickelt am Fraunhofer-Institut in Dortmund und WHITNESS, eine amerikanische Entwicklung, sind zwei datengesteuerte Baustein-Simulatoren mit integrierten Steuerungsstrategien für die Transportbausteine. Sie verfügen über eine beeindruckende graphische Bedienoberfläche, die graphischen Modellaufbau, Experimentieren, Animation und Ergebnispräsentation im gleichen Layout ermöglicht.

Die Vorgabe eines Sortiments an Bausteinen läßt keine speziellen Maschinentypen zu. Es sind aber gerade die Spezialmaschinen, an deren Verhalten man besonders interessiert ist, da sie sehr teuer sind, meist eine zentrale Rolle im Fertigungsablauf spielen und mit ihnen keine Betriebserfahrung vorliegt.

In erster Linie liegen ihre Schwächen aber darin, daß individuelle Steuerungsstrategien – wenn überhaupt – nur mit genauen Kenntnissen des Systems realisierbar sind. Für auftragsorientierte (z.B. flexible) Fertigungsanlagen ist dieser Punkt jedoch entscheidend. Wie im Abschnitt 1.5.1 dargelegt wurde, erhält man die interessierenden Leistungsgrößen aus der Verfolgung der Aufträge, d.h. aus der Kommunikation der Leit- und Steuerrechner und nicht aus dem Layout des Maschinenparks. Für diese Betrachtungsweise sind die gängigen Baustein-Simulatoren nicht ausgelegt, da sie vor allem Maschinen, Transporteinrichtungen und Läger als Bausteine zur Verfügung stellen.

SIMPRO ist ein datengesteuerter Simulator, der auf der Darstellung von Petri-Netzen beruht, die zu diesem Zweck um neue Elemente ergänzt wurden. Um Steueralgorithmen nicht durch ein unanschauliches Programm beschreiben zu müssen, wählte man Petri-Netze als Beschreibungsmittel. Petri-Netze haben zudem den Vorteil, daß sie hierarchisch in Komponenten zerlegbar sind. Eine Anlage kann demnach Zug um Zug am Bildschirm zusammengestellt werden. Der Ablauf in Petri-Netzen ist auch animierbar, so daß sowohl der Fluß der bewegten Teile als auch Zustandsübergänge in der Steuerung am Bildschirm beobachtet werden können. Auch hier kann das Layout der Modellerstellung unmittelbar für die Animation herangezogen werden.

Diesen unzweifelhaften Vorteilen stehen leider auch beträchtliche Nachteile gegenüber: Das gesamte Modell ist in Form eines Petri-Netzes abzulegen. Da bereits einzelne Teilnetze eine enorme Größe annehmen, geht schnell die Übersicht verloren. Will man die Information nicht nur auf dem Bildschirm, sondern auch auf Papier haben, dann sind riesige Pläne auszugeben. Viele Auswahlstrategien lassen sich allein mit Petri-Netzen nur sehr umständlich

beschreiben. Bereits beim Suchen in einer Liste nach bestimmten Kriterien entstehen nahezu unüberwindliche Schwierigkeiten. Ein Fertigungstechniker ist sicherlich nicht in der Lage, Steuerungen mit Hilfe von Petri-Netzen zu beschreiben. Wenn aber schon ein Informatiker hinzugezogen werden muß, dann ist nicht einzusehen, warum dann nicht eine kompaktere Beschreibung gewählt werden sollte.

Durch die vielen Erweiterungen, die eingeführt wurden, um ein Petri-Netz-Modell für fertigungstechnische Anwendungen einsatzfähig zu machen, ist eine ganz neue Modellwelt entstanden, die mit der ursprünglichen Intention der Petri-Netze letztlich nicht mehr viel gemein hat.

SIMPLEX-II, das vierte Projekt, wurde am Institut für Mathematische Maschinen und Datenverarbeitung der Universität Erlangen durchgeführt. Bei diesem Projekt ging man von vornherein davon aus, daß die gesteckten Ziele nicht in einem Schritt zu verwirklichen sind.

So einfach es ist, datengesteuerte Simulatoren mit einer komfortablen Oberfläche auszustatten, so mühsam ist es aber auch, dieses Konzept um neue Bausteine zu erweitern, wenn die angebotenen Komponenten für den Anwendungsfall nicht ausreichen. Nur der Entwickler des Simulators selbst ist hierzu in der Lage. Jeder neue Baustein bedeutet aber eine Erweiterung des originalen Simulators. Um nicht eine Flut von Spezialbausteinen führen oder für jeden Anwender eine eigene Version unterhalten zu müssen, wird der Entwickler aber nur bereit sein, solche Bausteine neu aufzunehmen, die von allgemeinem Interesse sind. Der Anwender wird daher in der Regel mit der angebotenen Modellwelt vorlieb nehmen müssen.

Daher entstand die Idee, das gesteckte Ziel zweistufig zu erreichen. In der ersten Stufe sollte ein Werkzeug zur Verfügung gestellt werden, mit dessen Hilfe Bausteine ohne Kenntnis eines darunterliegenden Simulationspakets definiert werden können. In der zweiten Stufe sollte eine graphische Bedienoberfläche entstehen, die nicht nur mit vorgefertigten Bausteinen operieren kann, sondern die es gleichwohl erlaubt, neue Bilder (Icons) für neue Komponenten zu gestalten und diese wiederum für den Modellaufbau heranzuziehen. Darüber hinaus sollte es möglich sein, Icons für eine Animation vorzubereiten.

Auf diese Weise sollte die gleiche komfortable Bedienung erreicht werden wie man sie von den datengesteuerten Simulatoren kannte, allerdings mit einer viel höheren Flexibilität, da der Anwender nicht nur mit einem vordefinierten Satz von Komponenten auskommen mußte.

Aber auch bei dieser Vorgehensweise war klar, daß der fertigungstechnische Planer Erweiterungen an seinem Simulationssystem wegen des hohen Einarbeitungsaufwands in der Regel nicht selbst vornehmen kann. Im Unterschied zu den datengesteuerten Baustein-Simulatoren sind aber keine Eingriffe in die Simulationssoftware notwendig und daher können Erweiterungen auch von anderen qualifizierten Personen durchgeführt werden. Dienstleistungserbringer wie OR-Abteilungen im eigenen Betrieb, Unternehmensberatungen oder Ingenieurbüros wären in der Lage, diese Aufgabe übernehmen. Auf diese Weise könnte sich jeder Anwender die für ihn relevanten Bausteine erstellen und in einer Modellbank ablegen lassen.

Die Forderung nach modularer Modellerstellung soll es demnach auch möglich machen, Modelle graphisch darzustellen bzw. am Bildschirm einzugeben. Für die Verwendbarkeit einer Modellbeschreibungssprache als Grundlage für Baustein-Simulatoren ist dieser Gesichtspunkt von ausschlaggebender Bedeutung. Die Modularisierung von Modellen wird daher ein zentrales Thema dieser Arbeit sein.

# Kapitel 2

# Sprachentwurf im Sinne der klassischen Systemtheorie

In diesem Kapitel sollen die Grundlagen der systemtheoretischen Modellbeschreibung erläutert werden. Es handelt sich hierbei um eine spezielle Form der zustands-orientierten Betrachtungsweise, wie sie in den Ingenieurwissenschaften üblich ist. Wir wollen darstellen, daß diese Betrachtungsweise als Basis für eine deklarative Sprache zur Beschreibung von Modellen dienen kann, und die Wesenszüge und Vorteile einer solchen Sprache herausarbeiten.

Um ein tieferes Verständnis zu erzielen, wird auch eine konkrete Sprachsyntax vorgestellt, damit sich der Leser anhand von Beispielen mit der Materie auseinandersetzen kann. Dieser Sprachentwurf wird dann in den folgenden Kapiteln weiterentwickelt, um dem Ziel, Warteschlangen- und Transportmodelle zu formulieren, näherzukommen.

## 2.1 Systemtheoretische Modellbeschreibung

### 2.1.1 Die zustands-orientierte Sichtweise

Von einem Simulationsprogramm erwartet man, daß es (innerhalb einer gewissen Genauigkeit) Beobachtungen ermöglicht, die denen am realen System entsprechen.

Jede Modellbildung stützt sich daher direkt oder indirekt (bei Anwendung einer Theorie) auf Beobachtungsdaten, die an einem realen System durch Experimente gewonnen wurden. Ein Modell mathematisch-formal zu beschreiben (spezifizieren) heißt, Zusammenhänge (Gesetzmäßigkeiten) zwischen diesen Daten herauszufinden, die stets, d.h. zu jedem beliebigen Zeitpunkt, Gültigkeit besitzen. Bei gleichen Anfangsbedingungen kann dann ein Simulationsprogramm – unter Anwendung dieser Gesetzmäßigkeiten – diese Daten reproduzieren.

Während wir ein Experiment durchführen, beobachten wir unser zu untersuchendes System an vorher ausgesuchten, wohlüberlegten Stellen. Über eine gewisse Zeit hinweg zeichnen wir an jedem Beobachtungsort eine oder mehrere Eigenschaften des Systems auf.

Eine objektivierte Beobachtung, d.h. eine Beobachtung die für alle Betrachter gleich ist, bezeichnen wir als Messung. Eine Messung wird durch Vergleich mit einem Normal durchgeführt und wählt aus einer definierten Menge möglicher Werte (Attributausprägungen) einen bestimmten Wert aus.

Eine Beobachtung

$$v: \quad \mathcal{T} \rightarrow V$$

läßt sich demnach mathematisch als eine Funktion der Zeit ansehen. Die Werte, welche die Beobachtung v annehmen kann, gehören einer Eigenschaftsmenge $V$ an. Die Werte, welche die Systemzeit $t$ annehmen kann, sind einer Zeitmenge $\mathcal{T}$ entnommen.

Wir gehen nun von der Annahme aus, daß es möglich ist, zum gleichen Zeitpunkt an verschiedenen Stellen des Systems Beobachtungen auszuführen. Um zeitliche Veränderungen festzuhalten, soll es uns genügen, Beobachtungen in endlichen Zeitabständen, also nicht etwa kontinuierlich, anzustellen. Aus dem Abtasttheorem geht hervor, daß uns bei genügend klein gewählten Zeitabständen keine Information verloren geht.

Ohnehin besitzt jeder reale Beobachter (Meßgerät) nur ein begrenztes zeitliches Auflösungsvermögen. Es ist daher absolut korrekt, die Zeitmenge der Systemzeit (nicht der Modellzeit !) stets als diskret und unendlich abzählbar anzusehen:

$$\mathcal{T} \quad = \quad t_0, \; t_1, \; t_2, \quad \ldots \; , \; t_k, \quad \ldots$$

Weiterhin gehen wir davon aus, daß es möglich ist, alle Beobachtungen gerade zu diesen Zeitpunkten auszuführen. Die Zusammenfassung aller Beobachtungen am realen System zu einem Zeitpunkt bezeichnen wir als den Systemzustand.

$$V(t_k) \quad := \quad [ \; v_1(t_k) \quad v_2(t_k) \quad \ldots \quad v_i(t_k) \quad \ldots \; ]$$

Alle über die Zeit hinweg am System gewonnenen Meßdaten lassen sich nun in einer Tabelle anordnen, die folgende Gestalt hat:

| Zeit | Beobachtungen | | | |
|---|---|---|---|---|
| $t$ | $v_1$ | $v_2$ | $v_3$ | $v_4$ |
| $t_0$ | | | | |
| $t_1$ | | | | |
| $t_2$ | | | | |
| $t_3$ | | | | |
| ... | | | | |
| $t_k$ | | | | |
| ... | | | | |
| ... | | | | |

Als zustands-orientierte Beschreibung bezeichnen wir schließlich einen Satz von Gleichungen, der die Beziehungen zwischen den Meßwerten in dieser Tabelle herstellt. Im Sinne der Aussagenlogik handelt es sich bei den Gleichungen um allgemeingültige Aussageformen.

### 2.1.2 Aufstellen der mathematische Beziehungen

Mathematische Beziehungen lassen sich unterscheiden in:

| | | | |
|---|---|---|---|
| – statische | Beziehungen: | $G\ (V(t_k))$ | $= 0$ |
| – dynamische | Beziehungen: | $F\ (V(t),\ V(t_{k+1}))$ | $= 0$ |
| – unabhängige | Beziehungen: | $E\ (v_i(t_k))$ | $= 0$ |

Statische Beziehungen verknüpfen Daten eines Zeitpunkts (Zustands) miteinander. Dynamische Beziehungen verknüpfen Daten zweier aufeinanderfolgender Zeitpunkte miteinander. Wenn Totzeiten eine Rolle spielen, werden auch weiter zurückliegende Zustände mit einbezogen. Unabhängige Beziehungen geben einzelne Größen als reine Zeitfunktion, d.h. unabhängig vom jeweiligen Modellzustand an.

Ein Satz von mathematischen Beziehungen weist jedoch nur dann eine einwandfreie Semantik auf, wenn sich aus den angegebenen Beziehungen eine eindeutige Aussage über den Zeitverlauf jeder Größe treffen läßt.

Hierzu müssen für ein Gleichungssystem mit $M$ Beziehungen und $N$ Größen die folgenden Forderungen erfüllt sein:

1) Die Beziehungen müssen widerspruchsfrei sein.

Beispiel: Es seien die drei Gleichungen

$$\begin{aligned} x + 2y + z &= 0; \\ x - 2y - z &= 0; \\ x &= \sin(\omega t); \end{aligned}$$

mit den Größen $x$, $y$ und $z$ angegeben.

Aus den ersten beiden Gleichungen folgt die Aussage $x = 0$, der die letzte Gleichung offensichtlich widerspricht.

2) Der Satz von Beziehungen muß vollständig sein. Wenn das Modell $N$ Modellgrößen enthält, müssen mindestens $M \geq N$ Beziehungen angegeben sein, wobei $N$ Beziehungen voneinander unabhängig sein müssen, d.h. von den $M$ angegebenen Beziehungen dürfen nicht mehr als die überzähligen $M - N$ Beziehungen aus den übrigen Beziehungen abgeleitet werden können.

Beispiel: Es seien die drei Gleichungen

$$\begin{aligned} x + 4y + z &= 0; \\ x + 2y - z &= 0; \\ y + z &= 0; \end{aligned}$$

mit den Größen $x$, $y$ und $z$ angegeben.

Die beiden ersten Beziehungen implizieren die dritte, d.h. das Gleichungssystem ist (noch) nicht vollständig, man sagt auch, es ist unterbestimmt.

In aller Regel ist $M = N$, d.h. man gibt nicht mehr Gleichungen an als Größen vorhanden sind. In diesem Fall müssen alle Beziehungen voneinander unabhängig sein, d.h. es darf dann keine einzige Gleichung aus den übrigen abgeleitet werden können.

Eine weitere Forderung ergibt sich aus der Tatsache, daß eine Größe zu einem Zeitpunkt stets nur einen einzigen Wert annehmen kann.

3) Jede Modellgröße muß eindeutig bestimmt sein.

Beispiel: Für die Größe $X$ sei die Gleichung

$$(X-2)^2 = T$$

gegeben.
Diese Gleichung liefert zwei Lösungen, nämlich

$$X_1 = 2+\sqrt{T}$$
$$X_2 = 2-\sqrt{T}$$

Der Wert der Größe X ist durch die angegebene Gleichung nicht eindeutig bestimmt.

Um in solchen Fällen die Eindeutigkeit sicherzustellen, sind Zusatzbedingungen erforderlich, welche alle nicht gültigen Lösungen ausschließen.

Einen Satz von Beziehungen (ggf. mit Zusatzbedingungen) bezeichnen wir als eine formale Modellbeschreibung oder auch als mathematisches Modell. Eine Modellbeschreibung enthält die gesamte Information, die man benötigt, um aus einem gegebenen Anfangszustand die Systemdaten zu reproduzieren. Aussagenlogisch betrachtet, werden dabei die Ausageformen zu konkreten Aussagen.

Lassen sich die Gleichungen nach den Modellgrößen explizit auflösen, sprechen wir von einer expliziten Modelldarstellung, andernfalls von einer impliziten Modelldarstellung.

### 2.1.3 Implizite Modelldarstellungen

Wir haben festgestellt, daß jede Systemgröße einen Funktionsverlauf über der Zeit repräsentiert und damit zu jedem Zeitpunkt einen eindeutigen Wert annimmt. Naturgemäß ist zu einem Zeitpunkt auch nur eine Beobachtung möglich. Wie ist es dann überhaupt denkbar, daß Gleichungen mit mehreren Lösungen in eine Modellbeschreibung gelangen ?

Die Ursache liegt darin, daß Modellgleichungen vielfach aus allgemeinen Beziehungen einer Theorie abgeleitet werden. Solche allgemeinen Beziehungen sind Bilanzgleichungen oder Gleichgewichtsbeziehungen.

Bilanzgleichungen besagen, daß eine Größe innerhalb eines abgeschlossenen Gebietes konstant bleibt.

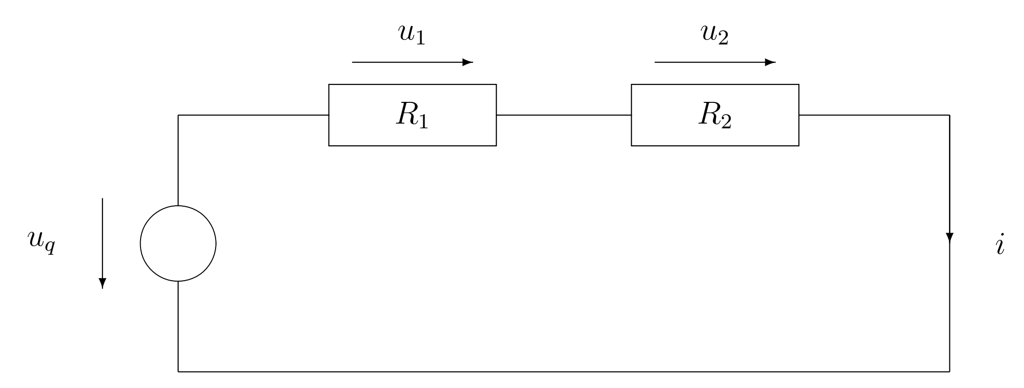
Beispiel: Die Summe aller Energien, Massen, Stoffmengen, Impulse und Drehimpulse bleibt erhalten.

Gleichgewichtsbeziehungen besagen, daß sich gleichartige Größen, die an einem gemeinsamen Punkt wirken, zu Null summieren.

Beispiel: Die Summe aller Kräfte, Drehmomente, Ströme etc. in einem gemeinsamen Punkt ist stets null.

Werden nun Zusammenhänge zwischen beteiligten Größen durch nichtlineare Funktionen beschrieben, dann erhält man Gleichungen, die nicht explizit nach allen Modellgrößen auflösbar sind.

Beispiel: Reihenschaltung zweier Widerstände




Einer der Widerstände zeigt lineares Verhalten, der andere wird durch ein Polynom 3. Grades beschrieben. Für die unbekannten Größen $U_q$, $U_1$, $U_2$ und $I$ gelten die folgenden Beziehungen:

Bauelementegleichungen:

$$\begin{aligned} U_q &= U_0 = const; \\ U_1 &= R_1 \cdot I; \\ U_2 &= R_{21} \cdot I + R_{22} \cdot I^2 + R_{23} \cdot I^3 \end{aligned}$$

Maschengleichung (Gleichgewichtsbeziehung):

$$U_q = U_1 + U_2;$$

Dieses Gleichungssystem ist nicht explizit nach $I$ auflösbar.

Im Falle diskreter Modelle spielen Bilanzgleichungen und Gleichgewichtsbeziehungen aber keine Rolle. Hier ist es in der Regel immer möglich, eine explizite Modelldarstellung anzugeben.

Da wir uns hier in erster Linie mit der Beschreibung diskreter Modelle befassen wollen, gehen wir im folgenden von der Voraussetzung aus, daß sich die Modellbeziehungen explizit darstellen lassen.

Diese Einschränkung führt uns zum Konzept der systemtheoretischen Modellbeschreibung und erleichtert es uns ganz wesentlich, die gestellten Anforderungen an die Modellbeschreibungssprache zu erfüllen.

### 2.1.4 Die explizite Modelldarstellung der Systemtheorie

Wenn wir in der Lage sind, jede einzelne Modellgröße explizit auszudrücken, ergibt sich für jede Modellgröße eine Definitionsgleichung. Je nachdem, welche Art von Beziehung eine Größe zum Ausdruck bringt, unterscheidet man die Modellgrößen $v_i$ in:

- Eingangsgrößen: $x_i(t_k)$
- Zustandsgrößen: $z_i(t_{k+1}) \ := \ f_i\ (X(t_k),\ Z(t_k),\ t_k);$
- Ausgangsgrößen: $y_i(t_k) \ := \ g_i\ (X(t_k),\ Z(t_k),\ t_k);$

Der Operator ':=' drückt aus, daß die Variable auf der linken Seite durch den Ausdruck auf den rechten Seite zu jedem Zeitpunkt definiert wird.

Die Namensgebung beruht auf folgender Vorstellung:

Eingangsgrößen sind Größen, auf die das System selbst keinen Einfluß hat. Man kann sie auch als unabhängige Umgebungsgrößen oder als Sensorgrößen bezeichnen, weil das System die Werte dieser Größen aus seiner Umgebung bezieht.

Ausgangsgrößen sind Größen, die von der Umgebung eines Systems beobachtet werden können. Sie hängen von den aktuellen Werten der Eingangs- und Zustandsgrößen ab und können daher auch als abhängige Größen bezeichnet werden. Da ihr Wert ohne jeden Zeitverzug den Eingangs- und Zustandsgrößen folgt, könnte man sie auch Spontangrößen nennen. Jede Ausgangsgröße $y_i$ wird durch eine Ausgabefunktion $g_i$ beschrieben.

Ein System ohne Zustandsgrößen zeigt auf gleiche Eingangsbelegung stets die gleiche Reaktion am Ausgang. Erhält man nicht immer die gleichen Werte am Ausgang, macht man hierfür den aktuellen "Zustand" des Systems verantwortlich. Der aktuelle Zustand wiederum stellt sich als Folge eines Anfangszustands und des bisherigen Verlaufs der Eingangsvariablen ein. Für die Änderung eines Zustands ist eine mehr oder weniger große Zeitspanne erforderlich. Der Folgezustand zum Zeitpunkt $t(t_{k+1})$ wird für jede Zustandsgröße $z_i$ mit Hilfe einer lokalen Zustandsüberführungsfunktion $f_i$ berechnet.




Die Systemtheorie [Pich 75] definiert die Zustandsdarstellung eines System S formal als ein 11-Tupel $(A, B, X, Y, Q_0, Q, Z, F, G, \mathcal{T}, t_0)$ mit

- der Eingangszeitfunktion $X(t)$, der Ausgangszeitfunktion $Y(t)$, der Zustandszeitfunktion $Z(t)$ und den dazugehörigen Wertemengen $A$, $B$ und $Q$
- der Zeitmenge $\mathcal{T} = \{t_0,\ t_1,\ \ldots\}$, deren Elemente in aufsteigender Ordnung vorliegen
- dem Anfangszustand $Q_0$ zum Zeitpunkt $t_0$, d.h. $Z(t_0) = Q_0$
- der lokalen Überführungsfunktion $F$, wobei gilt:
  $Z(t_{k+1}) \ := \ F(X(t_k),\ Z(t_k),\ t_k)$
- der Ausgabefunktion $G$, wobei gilt:
  $Y(t_k) \ := \ G(X(t_k),\ Z(t_k),\ t_k)$

Dieses Schema entspricht sehr gut der Anlage von Experimenten: das System als unbekannter Wirklichkeitsausschnitt, die Ausgangsvariablen als die für die Fragestellung interessierenden

Größen und die Eingangsvariablen als diejenigen Variablen, von denen man einen Einfluß auf die Ausgangsgrößen erwartet.

Stellt man sicher, daß jede Zustandsgröße und jede Ausgangsgröße durch genau eine Bestimmungsgleichung definiert wird, dann sind die Beziehungen zwangsläufig vollständig und widerspruchsfrei. Die Unabhängigkeit wird dadurch erreicht, daß die Bestimmungsgrößen $Z(t_{k+1})$ und $Y(t_k)$ auf der rechten Gleichungsseite nicht vorkommen dürfen.

Eine korrekte Modellsemantik läßt sich leider nicht mehr so einfach sicherstellen, wenn wir Systeme zu einem größeren Modell zusammenfügen.

### 2.1.5 Zusammenfügen von Systemen

Die (zunächst unbestimmten) Eingangsvariablen ermöglichen es, größere Modelle aus Teilsystemen zusammenzusetzen. Den Zeitverlauf einer Eingangsvariablen erhält man, indem man diese mit einer Ausgangsvariablen eines anderen Systems gleichsetzt. Sind alle Eingangsvariablen mit Ausgangsvariablen verbunden, ist das gesamte Modell abgeschlossen.

Beispiel: Das System $S1$ enthalte die Gleichungen

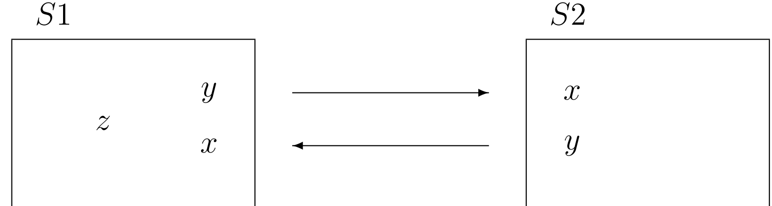
$$\begin{aligned} z(t_{k+1}) &:= f(x, z); \\ y(t_k) &:= g(z); \end{aligned}$$

und das System $S2$ die Gleichung

$$y(t_k) := g(x);$$

Wir stellen nun die folgenden Verbindungen her:

$$\begin{aligned} S2.x &\equiv S1.y; \\ S1.x &\equiv S2.y; \end{aligned}$$




Dadurch erhalten wir ein abgeschlossenes Modell.

Durch das Gleichsetzen der Eingangsvariablen mit Ausgangsvariablen sind aus den Gleichungen des Gesamtmodells die Eingangsvariablen eliminiert.

Beispiel: Die Gleichungen für das Gesamtmodell lauten:

$$\begin{aligned} S1.z(t_{k+1}) &:= f(S1.z,\ S2.y); \\ S1.y(t_k) &:= g_1(S1.z); \\ S2.y(t_k) &:= g_2(S1.y); \end{aligned}$$

Es fällt auf, daß nun sowohl in der Zustandsüberführungsfunktion als auch in einer Ausgabefunktion Ausgangsvariablen auftreten. Ob das Modell korrekt ist, ist nach einem Zusammensetzen aus Teilsystemen nicht mehr ohne weiteres sichergestellt.

Beispiel: Wenn man den Eingang des Systems $S2$ mit dem Ausgang des gleichen Systems verbindet, d.h. $S2.x \equiv S2.y$ , dann erhält man keine korrekte Modellsemantik mehr.

$$S2.y(t_k) \ := \ g_2(S2.y);$$

Eine Modellbeschreibung wird praktikabler, wenn man die strenge systemtheoretische Darstellung etwas abändert. Da sie die Korrektheit des Modells ohnehin nicht sicherstellen kann, sei es deshalb erlaubt, einige Modifikationen vorzunehmen.

### 2.1.6 Modifizierte Zustandsdarstellung

Für praktische Anwendungen ist das vorgestellte Schema in drei Punkten etwas zu einschränkend:

1. Ausgangsvariablen dürfen weder zur Berechnung des neuen Zustands noch zur Bestimmung weiterer Ausgangsvariablen herangezogen werden.
2. Zusammengesetzte Modelle lassen sich nur durch die Verbindung von Ausgangsvariablen mit Eingangsvariablen erzeugen. Die Werte von Zustandsvariablen können nur über eine zusätzliche Ausgangsvariable als Zwischengröße nach außen gegeben werden.
3. Um die Eingangsvariablen zu belegen, ist stets ein weiteres System erforderlich. Ein System, das Eingangsvariablen besitzt, kann nicht einmal zu Testzwecken für sich alleine betrieben werden.

Die Tatsache, daß Ausgangsvariablen weder in der Zustandsüberführungsfunktion noch in der Ausgabefunktion verwendet werden dürfen, stellt zwar aus mathematischer Sicht keine Einschränkung dar, weil für eine Ausgangsgröße jederzeit ihre Definitionsgleichung eingesetzt werden kann, aus praktischer Sicht ist diese Einschränkung jedoch erheblich.

Beispiel: Die Gleichungen

$$\begin{aligned} z(t_{k+1}) &:= y_1 + y_2; \\ y_1(t_k) &:= z \cdot x + 3; \\ y_2(t_k) &:= z + y_1; \\ y_3(t_k) &:= y_1 \cdot y_2; \end{aligned}$$

müßten lauten:

$$\begin{aligned} z(t_{k+1}) &:= y_1 + y_2; \\ y_1(t_k) &:= z \cdot x + 3; \\ y_2(t_k) &:= z + z \cdot x + 3; \\ y_3(t_k) &:= (z \cdot x + 3) \cdot (z + z \cdot x + 3); \end{aligned}$$

Die Ausgabefunktionen können sehr komplex werden und wiederholen dabei Terme, die für andere Ausgangsvariablen bereits berechnet wurden.

Im folgenden arbeiten wir nur noch mit der modifizierten Modelldarstellung. Sie ist gekennzeichnet durch Beziehungen der Form:

- Sensorvariablen: $x_i(t_k) \ := \ e_i\ (t_k);$
- Zustandsvariablen: $z_i(t_{k+1}) \ := \ f_i\ (X(t_k),\ Y(t_k),\ Z(t_k),\ t_k);$
- abhängige Variablen: $y_i(t_k) \ := \ g_i\ (X(t_k),\ Y(t_k),\ Z(t_k),\ t_k);$

Eingangsgrößen heißen fortan nur noch Sensorvariablen. Ihnen wird eine Zeitfunktion (Sensorfunktion) zugeordnet, wenn die Größe nicht angeschlossen ist.

Ausgangsgrößen erhalten eine etwas andere Bedeutung, da sie nun auch als Zwischengrößen fungieren können und nicht mehr die einzigen Größen sind, die nach außen gegeben werden können. Aus diesem Grund wollen wir in Zukunft von abhängigen Variablen sprechen.

Abhängige Variablen dürfen sowohl zur Berechnung von Zustandsvariablen als auch von anderen abhängigen Variablen verwendet werden. Ihre Definitionsgleichungen bezeichen wir als statische Beziehungen oder algebraische Gleichungen, die Funktion zu ihrer Berechnung entsprechend als algebraische Funktion.

Zustandsvariablen werden durch Zustandsüberführungsfunktionen definiert und die Definitionsgleichungen zu ihrer Berechnung als dynamische Beziehungen bezeichnet.

Weiterhin wollen wir zulassen, daß auch Zustandsvariablen mit Sensorvariablen verbunden werden dürfen.

Aus der Verwendung von abhängigen Variablen in Zustandsüberführungsfunktionen ergeben sich überhaupt keine Probleme, denn der zu berechnende neue Zustand (zum Zeitpunkt $t_{k+1}$) geht aus dem bereits bekannten aktuellen Zustand hervor.

$$\text{Zustandsvariablen:} \quad z_i(t_{k+1}) \ := \ f_i\ (X(t_k),\ Y(t_k),\ Z(t_k),\ t_k);$$

Bei den algebraischen Gleichungen ist dies etwas anders, da sich damit u.U. keine expliziten Beziehungen mehr ergeben.

Beispiele:

a) $y \ := \ (x+z)/y;$

b) $y_1 \ := \ z + y_2;$
$y_2 \ := \ x + y_1;$

Wie wir aber gesehen haben, kann dieser Effekt auch bei einer unglücklichen Verschaltung einzelner Systeme auftreten. Aus diesem Grund müssen wir uns ohnehin überlegen, auf welche Weise die semantische Korrektheit bzw. der explizite Charakter der Modellbeschreibung bewahrt werden kann.

Bei einem korrekt beschriebenen Modell muß es ganz offensichtlich möglich sein, die abhängigen Variablen in eine solche Reihenfolge zu bringen, daß sich die zu definierende abhängige Variable nur aus Eingangs- und Zustandsgrößen und bereits vorher definierten abhängigen Variablen errechnet.

$$\text{abhängige Variablen:} \quad y_i(t_k) \ := \ g_i\ (X(t_k),\ y_1(t_k),\ y_2(t_k),\ \ldots\ ,\ y_{i-1}(t_k),\ Z(t_k),\ t_k);$$

Wie man diese Eigenschaft an den Modellgleichungen erkennen kann, zeigt der folgende Abschnitt.

## 2.2 Semantische Korrektheit

Existiert – so wie dies bei der expliziten Modelldarstellung der Fall ist – für jede Modellgröße eine Bestimmungsgleichung, dann lassen sich die Beziehungen zwischen den Größen durch einen Abhängigkeitsgraphen zum Ausdruck bringen.

Wir betrachten hier zunächst den Fall, daß das Modell nicht in mehrere Systeme (Komponenten) zerlegt ist.

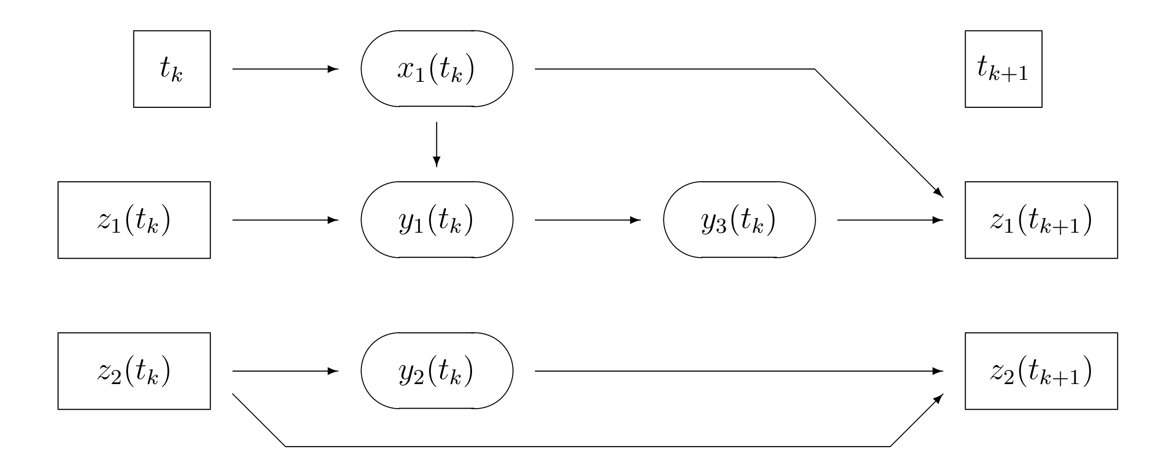
Die Knoten des Graphen repräsentieren die nicht verbundenen Sensorvariablen $x_i(t_k)$, die abhängigen Variablen $y_i(t_k)$ und die Zustandsvariablen $z_i(t_k)$ bzw. $z_i(t_{k+1})$, die Kanten stellen die funktionalen Abhängigkeiten dar.

Der Graph ist gerichtet und beginnt mit den Größen, die den Zustand zum Zeitpunkt $t_k$ kennzeichnen, d.h. mit den Zustandsvariablen $z_i(t_k)$ und der Zeit $t_k$, dargestellt in rechteckigen Kästchen. Er endet mit den Zustandsvariablen $z_i(t_{k+1})$ zum nächsten Zeitpunkt. Dazwischen liegen die nicht angeschlossenen Sensorvariablen $x_i(t_k)$ und die abhängigen Variablen $y_i(t_k)$, markiert durch Kreise.

Beispiel: Die funktionalen Abhängigkeiten der Modellgrößen

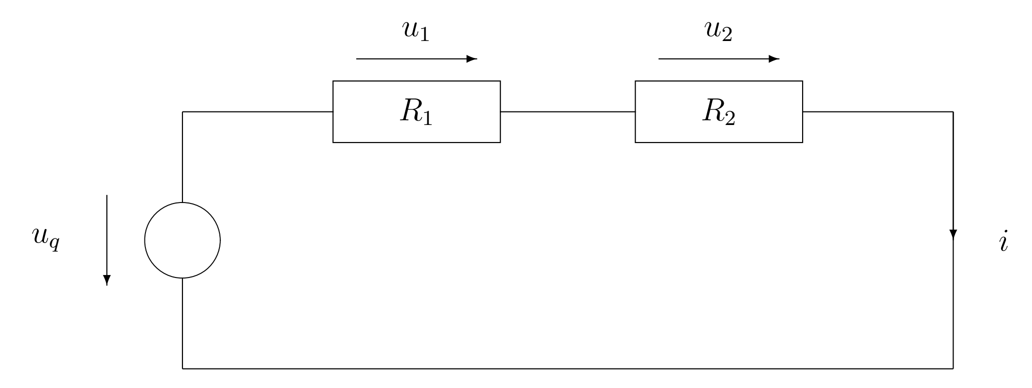
$$\begin{aligned}
x_1(t_k) &:= e_1(t_k);\\
y_1(t_k) &:= g_1(x_1, z_1);\\
y_2(t_k) &:= g_2(z_1, z_2);\\
y_3(t_k) &:= g_3(y_1);\\
z_1(t_{k+1}) &:= f_1(x_1, y_3);\\
z_2(t_{k+1}) &:= f_2(z_2, y_2);
\end{aligned}$$

lassen sich durch den folgenden Graphen darstellen:




Wenn nun in einer algebraischen Funktion abhängige Größen auftauchen, die noch nicht definiert wurden, dann bedeutet das, daß der Graph einen Rückwärtsverweis enthält. Mit anderen Worten: der Graph ist dann nicht mehr zyklenfrei.

Beispiel: Reihenschaltung zweier Widerstände



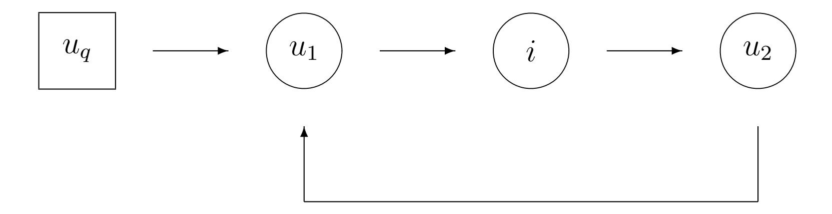

Die zunächst teilweise impliziten Modellgleichungen

$$
\begin{aligned}
u_1 &= R_1 \cdot i \\
u_2 &= R_2 \cdot i \\
u_q - u_1 - u_2 &= 0 \\
u_q &= \sin(t)
\end{aligned}
$$

lassen sich in das scheinbar explizite Gleichungssystem

$$
\begin{aligned}
i &= u_1/R_1 &&= i(u_1) \\
u_2 &= R_2 \cdot i &&= u_2(i) \\
u_1 &= u_2 - u_q &&= u_1(u_2,\ u_q) \\
u_q &= \sin(t)
\end{aligned}
$$

umformen. Der dazugehörige Abhängigkeitsgraph hat das folgende Aussehen:




Die vorgenommene Umformung ist zwar äquivalent, aber nicht zyklenfrei.

Der Abhängigkeitsgraph ermöglicht uns demnach die Überprüfung einer korrekten Semantik.

Es gilt daher:

Eine explizite Modellbeschreibung ist dann semantisch korrekt, wenn für jeden Modellzustand $(X(t),\ Y(t),\ Z(t),\ t)$ die folgenden Bedingungen erfüllt sind:

1. Für jede Modellgröße muß mindestens eine Definitionsgleichung angegeben sein.
2. Der durch die Definitionsgleichungen gegebene Abhängigkeitsgraph muß zyklenfrei sein.
3. Für jede Modellgröße darf höchstens eine Definitionsgleichung angegeben sein.

Die erste und zweite Bedingung sichern die Vollständigkeit der Modellbeschreibung, die dritte Bedingung stellt die Eindeutigkeit sicher und verhindert damit, daß weitere, eventuell widersprüchliche Aussagen hinzukommen.

Die erste und dritte Bedingung läuft natürlich darauf hinaus, daß für jede Modellgröße genau eine Definitionsgleichung anzugeben ist.

Beispiele: a) $Y_1 := Y_2;$
$Y_2 := Y_1;$

b) $Y_1 := 2Y_2;$
$Y_2 := 2Y_1;$

In beiden Fällen enthält der Abhängigkeitsgraph einen Zyklus.

Im ersten Fall sind die Gleichungen nicht unabhängig, da die erste Gleichung die zweite impliziert. Im zweiten Fall sind die beiden Gleichungen widersprüchlich.

Der hier eingeführte Abhängigkeitsgraph wird sich als allgemein nützliches Hilfsmittel für die Analyse zustandsorientierter Modelle erweisen. Neben der Prüfung auf korrekte Modellsemantik bringt er bei der Systemanalyse Klarheit über die Wirkungen der einzelnen Modellgrößen. Beim Entwurf eines Algorithmus zur Abarbeitung der Modellbeschreibung wird er uns dienlich sein, um Effizienz zu erreichen. Und im Kapitel über Modularisierung werden wir den Abhängigkeitsgraphen schließlich auf zusammengesetzte Modelle erweitern.

## 2.3 Spezielle Zustandsüberführungsfunktionen

Die Zustandsüberführungsfunktion (ZÜF) beschreibt, in welchen Zustand zum Folgezeitpunkt das Modell aus dem aktuellen Zustand übergeht.

Je nachdem, welche Zeitmenge man für die Modellzeit zugrunde legt, ergeben sich verschiedene spezielle Formen für die ZÜF. Da für das Verständnis der Beispielmodelle die Kenntnis der möglichen ZÜF erforderlich ist, werden diese im folgenden nochmals in aller Kürze vorgestellt.

Für die Kernaussagen dieser Arbeit ist jedoch die gewählte Zeitmenge und ZÜF ohne Belang. Alle Betrachtungen zur Semantik, zur Effizienz oder zur Modularisierung unseres Programmiermodells gelten für jede Art von ZÜF.

Unabhängig davon, mit welcher Zeitmenge wir arbeiten, wollen wir den Folgezustand $z_i(t_{k+1})$ abkürzend auch mit $\hat{z}_i$ und den Folgezeitpunkt $t_{k+1}$ abkürzend mit $\hat{t}$ bezeichnen.

Den aktuellen Modellzustand, der auch die Belegungen der Sensorvariablen und der abhängigen Variablen sowie die aktuelle Zeit enthält, kürzen wir mit $W(t)$ ab und bezeichnen ihn als erweiterten Modellzustand.

$$W(t) = (\ X(t)\ \ Y(t)\ \ Z(t)\ \ t\ )$$

Legt man für die Zeitmenge die Menge der nichtnegativen, ganzen Zahlen zugrunde, d.h. $\mathcal{T} = N_0$, dann erhält man die Darstellung der Automatentheorie.

$$z_i(t+1) \ := \ f\ (W(t));$$

Automatentheoretische Modelle können keine Aussage darüber machen, zu welchem realen Zeitpunkt (Systemzeit) ein bestimmter Zustand angenommen wird. Lediglich die Reihenfolge der Zustandsänderungen, d.h. der funktionelle Ablauf kann anhand eines solchen Modells studiert werden. Eine spezielle Darstellungsform für endliche Automaten sind die Petri-Netze.

Eine einfache Beschreibungsform für diskrete Modelle, bei der auch ein Bezug zur Systemzeit hergestellt wird, ergibt sich, wenn man für die Zeitmenge das Vielfache einer Zeiteinheit $\Delta t \in \mathcal{R}$ mit $\Delta t > 0$ wählt:

$$z_i(t+\Delta t) \ := \ f\ (W(t));$$

Nehmen dabei die ZÜF die spezielle Form

$$z_i(t+\Delta t) \ := \ z_i(t) \quad + \quad \Delta t \cdot r_i^{(\Delta t)}(W(t))$$

an, dann erhält man eine Modellbeschreibung mit Differenzengleichungen. Eine geeignete Notation, z.B.

$$\mathrm{DELTA}(\Delta t) z_i \ := \ r_i\ (W(t));$$

kann dafür Sorge tragen, daß nicht die gesamte ZÜF, sondern nur die Änderungsraten $r_i$ definiert werden müssen.
Differenzengleichungen kommen nur dann in Betracht, wenn im gesamten Modell mit dem stets gleichen Zeitinkrement gearbeitet werden kann und keine besonderen Anforderungen an die Genauigkeit gestellt werden.

In der Regel nimmt man die Modellzeit als reellwertig an. Man geht deshalb noch einen Schritt weiter und läßt als Zeitmenge reelle Zahlen und ein infitesimal kleines Zeitinkrement $dt \rightarrow 0$ zu. So erhält man eine Modellbeschreibung mit Differentialgleichungen und Ereignissen.

Ein diskreter Zustandsübergang wird durch die Ereignisfunktion

$$z_i(t+dt) \ := \ \hat{f}_i\ (W(t)) \qquad \text{bzw.}$$

$$\hat{z}_i \ := \ \hat{f}_i\ (W(t))$$

beschrieben.

Da $dt$ ein infinitesimal kleines Zeitinkrement darstellt, läßt sich ein realer Zeitfortschritt nur erzielen, wenn $\hat{f}_i$ eine diskrete Funktion mit folgender Eigenschaft ist:

$$\hat{f}_i\ (W(t)) \ = \ z_i(t) \qquad \text{falls} \quad W(t) \notin \mathcal{A}_i$$

Nur wenn eine Auslösebedingung erfüllt ist, d.h. der aktuelle Modellzustand der Auslösemenge $\mathcal{A}_i$ angehört, besitzt die Ereignisfunktion $\hat{f}_i$ einen Wert, der von seinem Vorgänger verschieden ist.

Die diskrete ZÜF nimmt daher stets die Form

$$z_i(t + dt) \ := \ \begin{cases} \hat{f}_i(W(t)) & \text{falls} \qquad W(t) \in \mathcal{A}_i \\ z_i(t) & \text{sonst} \end{cases}$$

an.

Da für die Formulierung eines Ereignisses nur der Fall interessant ist, bei dem sich die Zustandsvariable ändert, lassen wir auch die Kurzschreibweise

$$\hat{z}_i \ := \ \hat{f}_i\ (W(t)) \qquad \text{falls} \quad W(t) \in \mathcal{A}_i$$

zu.

Auf diese Weise lassen sich diskrete Funktionsverläufe lückenlos über der reellen Zeitachse beschreiben.

Eine weitere Voraussetzung hierfür ist, daß nach endlich vielen Zwischenzuständen kein weiteres Ereignis mehr stattfindet, d.h. die vereinigte Auslösemenge von allen Zustandsvariablen

$$\mathcal{A} \ = \ \bigcup_i \ \mathcal{A}_i$$

wieder verlassen wird. Erst dann wird wieder ein realer Zeitfortschritt möglich.

Nach einem realen Zeitfortschritt bleiben die Werte sämtlicher (diskreter) Zustandsvariablen konstant, bis eine Zeitbedingung dafür sorgt, daß wieder ein neues Ereignis stattfindet, d.h. eine Zustandsvariable ihren Wert sprunghaft ändert.

Beispiel: Anfangsbelegung: `z(0) = 0;   t_0 = 0;   DT = 1;`

Ereignis: `z^ :=  t   falls   t >=  z + DT`

Der Zeitverlauf von `z` wächst alle `DT` Zeiteinheiten um `DT`. Durch die Ausführung des Ereignis wird die Auslösebedingung unwahr. Erst wenn die Zeit den Wert `z + DT` erreicht, wird wieder ein Ereignis ausgeführt.

Um diskrete Zeitverläufe über einer reellen Modellzeit auch implementierungstechnisch korrekt zu behandeln, empfiehlt es sich, eine zweidimensionale Implementierungszeit einzuführen. Wie bereits in [Esch 90] ausführlich beschrieben, setzt sich ein Zeitpunkt aus zwei Komponenten, der realen Zeit $T$ und dem Moment $M$ (früher Takt) zusammen. Der Bezug zur Modellzeit wird durch die Beziehung

$$t \; = \; T + M \cdot dt \qquad \text{mit} \quad T \; = \; n \cdot R \quad \text{und} \quad M \in \mathcal{N}$$

hergestellt.

Dabei ist $T$ das Vielfache eines vorgegebenen Auflösungsvermögens $R$ (Gleitkommazahl) und $M$ ist ein Zähler, der angibt wieviele infinitesimal kleine Zeitinkremente verstrichen sind, d.h. wieviele Ereignisse zur realen Zeit $T$ bereits ausgeführt wurden.

Da die Zeit als Zahlenpaar $(T, M)$ geführt wird, kann dem neuen Zustand nach einem Ereignis eine eindeutige Zeitangabe zugeordnet werden.

Wir haben damit drei Zeitbegriffe kennengelernt:

- die Systemzeit, welche durch das Auflösungsvermögen der Uhren festgelegt ist
- die Modellzeit, welche die modellhafte Zeitvorstellung des Anwenders wiedergibt
- die Implementierungszeit, welche eine Annäherung an die Modellzeit wiedergibt

Bei Verwendung einer reellen Modellzeit lassen sich auch kontinuierliche Zustandsübergänge formulieren, welche durch die spezielle ZÜF

$$z_i(t + dt) \; := \; z_i(t) \; + \; f_i' \; (W(t)) \cdot dt$$

beschrieben werden. Die besondere Notation für diesen Fall beschränkt sich auf die Angabe der Ableitung, also auf die Differentialgleichung.

$$z_i' \; := \; f_i' \; (W(t));$$

Differentialgleichungen ermöglichen die Beschreibung stetiger Funktionsverläufe und sind nur auf reelle Modellvariablen anwendbar. Um auch stetige Funktionsverläufe mit Sprungstellen beschreiben zu können, ist die Einführung einer kombinierten ZÜF erforderlich.

Wie bereits in [Esch 90] beschrieben, läßt sich eine kombinierte ZÜF so definieren, daß reelle Modellvariablen sowohl durch Ereignisse als auch durch Differentialgleichngen beschrieben werden können.

```
z_i' :=  f_i' (W(t));
z_i^ :=  f_i^ (W(t));
```

Der Verlauf von reellen Zustandsgrößen mit reeller Zeitmenge wird demnach durch die Angabe von zwei Funktionen beschrieben:
Der Ableitungsfunktion $f'$ und der Ereignisfunktion $\hat{f}$.

Für das Verständnis der Beispiele in dieser Arbeit sollten diese Anmerkungen über spezielle ZÜF ausreichend sein.

## 2.4 Grundkonstruktionen des Sprachkonzepts

### 2.4.1 Definitionsgleichungen

Eine explizite Modellbeschreibung ist ein Satz von Definitionsgleichungen, d.h. von allgemeingültigen Aussageformen, für den Vollständigkeit und Eindeutigkeit zu gewährleistet ist.

Die Vollständigkeit können wir dadurch sicherstellen, indem wir für jede deklarierte Modellgröße mindestens eine Definitionsgleichung verlangen. Darüberhinaus fordern wir, daß der Abhängigkeitsgraph zyklenfrei ist.

Die Eindeutigkeit ergibt sich daraus, daß nicht mehr als eine Definitionsgleichung angegeben werden darf.

Unser Ziel ist es, einen Sprachentwurf anzugeben, der es möglich macht, diese Forderungen (durch einen Rechner) zu überprüfen, damit ein semantisch korrektes Modell sichergestellt ist. In diesem Abschnitt soll gezeigt werden, daß eine geeignete Syntax eine solche Überprüfung erleichtern kann.

Eine Definitionsgleichung bestimmt aus dem aktuellen Modellzustand den Wert einer Modellvariablen. Diese funktionale Abhängigkeit kann auf verschiedene Art und Weise dargestellt werden:

- durch explizite, geschlossene (algebraische) Ausdrücke mit oder ohne Fallunterscheidungen
- durch tabellarisch beschriebene Funktionen (ggf. mit Interpolation)
- durch algorithmisch beschriebene Funktionen

In der Mathematik sowie in allen höheren Programmiersprachen (FORTRAN, PASCAL, C, etc.) ist es üblich, Funktionen als Bestandteile von Ausdrücken zuzulassen. Wir beschränken daher die weiteren Ausführungen auf Ausdrücke mit und ohne Fallunterscheidung. Von den Funktionen verlangen wir eine derartige Realisierung, daß sie keinerlei Seiteneffekte auf die Argumente oder auf andere Modellvariablen ausüben können.

Wir setzen weiterhin voraus, daß der Ausdruck bzw. die Funktion für alle möglichen Zustände einen gültigen Wert liefert, d.h. keine Divisionen durch Null oder Verletzungen von Definitionsbereichen auftreten. Diese Fehlermöglichkeiten lassen sich leider erst während des Modellaufs, aber nicht bereits vom Übersetzer prüfen.

#### Definitionsgleichung ohne Fallunterscheidung

Hier gibt es keinen Grund von der in der Mathematik üblichen syntaktischen Notation abzuweichen.

```
assignment ::=   identifier [tr_spec]  ':='  expression ';'

tr_spec    ::=   " ' "  |  " ^ "
```

Der Übergangsspezifikator (`tr_spec`), der hinter dem zu definierenden Bezeichner steht, bestimmt, ob es sich

- um den Variablenwert zum aktuellen Zeitpunkt (kein Spezifikator)
- um den Variablenwert zum nachfolgenden Zeitpunkt (Spezifikator `" ^ "`) oder
- um die zeitliche Ableitung des Variablenwertes (Spezifikator `" ' "`)

handelt.

Durch den Spezifikator lassen sich algebraische Gleichungen, Ereignisse und Differentialgleichungen voneinander unterscheiden. Der besseren Lesbarkeit halber mag es aber angebracht erscheinen, Ereignisse und Differentialgleichungen, d.h. die Zustandsübergangsfunktionen, durch Schlüsselworte besonders hervorzuheben.

Beispiel:

```
DIFFERENTIAL EQUATIONS                               # freier Fall
   x' :=  v;
   v' := -g;
END

EVENT                                                # Reflexion am Boden
   v^ := -k*v  WHEN  x < 0 AND v < 0  END
END
```

**Definitionsgleichung mit Fallunterscheidung**

Die Forderung nach Vollständigkeit und Widerspruchsfreiheit muß auch bei Fallunterscheidungen gewährleistet sein. Eine Fallunterscheidung hat bekanntlich folgendes Aussehen:

$$u \;:=\; \begin{cases} f_1(W) & \text{falls} \quad W \in B_1 \\ f_2(W) & \text{falls} \quad W \in B_2 \\ f_3(W) & \text{falls} \quad W \in B_3 \\ \ldots\ldots & \text{falls} \quad \ldots\ldots \\ f_n(W) & \text{sonst} \end{cases}$$

Die Fallunterscheidung ist semantisch nur dann korrekt, wenn sich

a) die Mengen $B_1$, $B_2$, u.s.w. nicht überschneiden, d.h. wenn
$B_1 \cap B_2 \cap \ldots \cap B_{n-1} \;=\; \emptyset$ .
Nur dann ist die Eindeutigkeit gewährleistet.

b) die Mengen $B_1$, $B_2$, u.s.w. den gesamten Zustandsraum abdecken, d.h. wenn
$B_1 \cup B_2 \cup \ldots \cup B_{n-1} \;=\; \mathcal{Z} \times \mathcal{X} \times \mathcal{T}$.
Nur dann ist die Vollständigkeit gewährleistet.

Ist die erste Bedingung verletzt, ist in bestimmten Gebieten des Zustandsraums für die Variable $u$ mehr als ein Wert definiert. Wenn die zweite Bedingung verletzt ist, gibt es Gebiete des Zustandsraums, in denen für $u$ gar kein Wert definiert ist.

Durch Erzeugung eines geeigneten Codes lassen sich beide Fälle problemlos zur Laufzeit erkennen. Eine Fehlererkennung zur Übersetzungszeit ist jedoch nicht mehr mit vertretbarem Aufwand zu leisten und es ist nicht klar, ob es im allgemeinen Fall überhaupt lösbar ist. Dazu müßte es möglich sein, die einzelnen Bedingungen symbolisch zu verarbeiten.

Dieses Problem läßt sich dadurch umgehen, indem man erlaubt, daß die Zustandsraumzerlegung sukzessive erfolgt, d.h. ausgegrenzte Gebiete bei den weiteren Bedingungen nicht mehr berücksichtigt werden.

Die Schreibweise lautet dann:

$$u := \begin{cases} & f_1(V) & \text{falls} & W \in C_1 \\ \text{sonst} & f_2(V) & \text{falls} & W \in C_2 \\ \text{sonst} & f_3(V) & \text{falls} & W \in C_3 \\ \text{sonst} & \ldots\ldots & \text{falls} & \ldots\ldots \\ \text{sonst} & f_n(V) & & \end{cases}$$

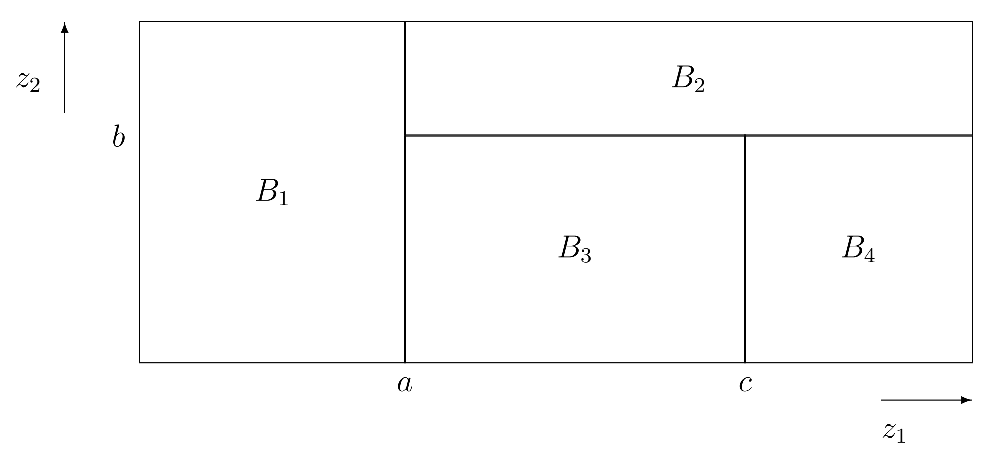
mit der Bedeutung:

$$u := \begin{cases} f_1(V) & \text{falls} & W \in C_1 & & \\ f_2(V) & \text{falls} & W \in C_2 & \text{und nicht} & (W \in C_1) \\ f_3(V) & \text{falls} & W \in C_3 & \text{und nicht} & (W \in C_1 \cup C_2) \\ \ldots\ldots & \text{falls} & \ldots\ldots & \text{und nicht} & (\ldots\ldots) \\ f_n(V) & \text{sonst} & & & \end{cases}$$

Die Mengen $C_1$, $C_2$ u.s.w dürfen bei dieser Schreibweise die bereits ausgegrenzten Gebiete mit umfassen.

$$C_i = B_i \cup B_{i-1} \cup \ldots \cup B_1$$

Die Fallunterscheidung in einem zweidimensionalen Zustandsraum kann man sich beispielsweise so vorstellen:




$$u := \begin{cases} & f_1(V) & \text{falls} & z_1 < a & \text{d.h.} & W \in B_1 \\ \text{sonst} & f_2(V) & \text{falls} & z_2 > b & \text{d.h.} & W \in B_2 \cup B_1 \\ \text{sonst} & f_3(V) & \text{falls} & z_1 < c & \text{d.h.} & W \in B_3 \cup B_2 \cup B_1 \\ \text{sonst} & f_n(V) & & & \text{d.h.} & W \in B_4 \cup B_3 \cup B_2 \cup B_1 \end{cases}$$

Diese Art der Formulierung hat auf jeden Fall den Vorteil, daß eine Unterteilung des Zustandsraumes übersichtlicher zu beschreiben ist.

Gleichzeitig bringt es den Effekt mit sich, daß die zu definierende Größe stets eindeutig definiert ist. Mehrdeutigkeiten schließt die Syntax von vornherein aus.

Diese Art von Pragmatik stieß in Diskussionen immer wieder auf Skepsis. Daher seien an dieser Stelle die wichtigsten Einwände und Gegenargumente zusammengestellt.

Die sonst in einer deklarativen Sprache vorzufindende Reihenfolgeunabhängigkeit der Aussageformen ist in diesem Fall scheinbar verletzt, weil die einzelnen Zweige der Fallunterscheidung nicht vertauschbar sind. Die Sprache weist an dieser Stelle vermeintlich gar prozedurale Züge auf.

Man muß sich jedoch darüber im klaren sein, daß nur die Definitionsgleichung als Ganzes, d.h. mit allen Zweigen, eine vollständige Aussageform darstellt. Die Definitionsgleichungen selbst lassen sich auch weiterhin in beliebiger Reihenfolge anordnen. Für den Ausdruck auf der rechten Gleichungsseite gelten ohnehin eigene Gesetze. So wird ein arithmetischer Ausdruck nach Prioritätsregeln (z.B. Punkt-vor-Strich) interpretiert. Auch hier ist eine bestimmte Abarbeitungsreihenfolge ausschlaggebend.

Ein anderer Einwand ist, daß die Bedingungen so unglücklich formuliert sein können, daß ein Zweig gar keine Berücksichtigung findet.

Beispiel:

$$u := \begin{cases} & f_1(V) \quad \text{falls} \quad z < 5 \\ \text{sonst} & f_2(V) \quad \text{falls} \quad z < 3 \\ \text{sonst} & f_3(V) \end{cases}$$

Der zweite Fall wird vom ersten vollständig mit eingeschlossen und kann daher nie zur Anwendung kommen.

Ist eine Bedingung gar allgemeingültig, d.h. für alle Zustände erfüllt, dann bleiben alle weiteren Zweige unberücksichtigt.

Die Modellbeschreibung ist in diesen Fällen zwar vollständig und eindeutig, dennoch ist klar, daß hier wohl ein gedanklicher Fehler bei der Formulierung der Bedingungen vorliegt, der nicht erkannt wird.

Dem sind zwei Dinge zu entgegnen: Erstens kann man prinzipiell nicht erwarten, daß sich gedankliche Fehler durch eine Fehlermeldung zur Laufzeit oder Übersetzungszeit äußern. In aller Regel – so auch hier – wird das Modell ein fehlerhaftes Verhalten zeigen. Zweitens ist die Formulierung der Bedingungen bei der ersten Variante ungleich komplizierter und läßt daher sehr viel häufiger Fehler erwarten.

In der Praxis hat sich gezeigt, daß die sukzessive Fallunterscheidung vom Anwender besser gehandhabt werden kann. Er wird durch die Syntax von Anfang an dazu angehalten, darüber nachzudenken, in welcher Weise der gesamte Zustandsraum zu unterteilen ist. Er muß nicht selbst für den gegenseitigen Ausschluß der Bedingungen sorgen und er kann eine Fallunterscheidung ohne große Mühe durch weitere Fälle ergänzen.

Wir stellen nun die Syntax für eine Fallunterscheidung in zwei verschiedenen Varianten vor: die Wertzuweisung mit bedingtem Ausdruck (WHEN-Konstrukt) und die bedingte Aussageform (IF-Konstrukt).

```
assignment ::=  identifier  [ tr_spec ]  ':='  conditional_expression

tr_spec ::=   ' ^ '
            | " ' "

conditional_expression ::=   expression ';'
                           | expression  WHEN  expression
                           { ELSE
                             expression  WHEN  expression }
                           [ ELSE
                             expression ]
                             END

conditional_statement ::=   IF     expression  LET  statement_sequence
                          { ELSIF  expression  LET  statement_sequence }
                          [ ELSE               LET  statement_sequence ]
                            END

statement ::=   assignment
              | conditional_statement

statement_sequence :=  statement  { statement }
```

Wie man erkennt, werden Aussageformen entweder durch einen Strichpunkt (einfache Wertzuordnung) oder durch das Schlüsselwort END (bedingte Aussageformen) abgeschlossen.

Die Ausdrücke hinter den Schlüsselwörtern WHEN und IF müssen selbstverständlich einen logischen Wert liefern.

Von diesen beiden Konstrukten ist im Grunde nur eines erforderlich. Dennoch haben je nach Anwendung beide Konstrukte ihre Berechtigung.

Das WHEN-Konstrukt hat seinen Vorteil darin, daß es sehr kompakt ist und deutlich macht, welche Größe durch die Aussageform definiert wird.

Das IF-Konstrukt hat den Vorteil, daß es mehrere Definitionsgleichungen aufnehmen kann, wenn gleiche Fälle zu unterscheiden sind. Ein weiterer Vorteil ist, daß es tiefere Schachtelungen des Zustandsraumes erlaubt.

Beispiel:

```
IF     x >  c  LET  y_1 :=  1;
                    y_2 :=  x;
               END
ELSIF  x < -c  LET  y_1 := -1;
                    y_2 := -x;
               END
ELSE           LET  y_1 :=  0;
                    y_2 :=  0;
               END
```

Dieses IF-Konstrukt definiert die beiden abhängigen Variablen `y_1` und `y_2`.

Für das WHEN-Konstrukt ist eine korrekte Semantik sehr leicht zu gewährleisten. Die Eindeutigkeit ist durch die sukzessive Fallunterscheidung erreicht worden und Vollständigkeit ist immer dann gegeben, wenn auf den ELSE-Zweig nicht verzichtet wurde.

Es wurde bereits erwähnt, daß man auf die Angabe des ELSE-Zweiges in Fallunterscheidungen verzichten kann, wenn ein Differentialquotient oder eine Ereignisfunktion definiert wird. Für diese Fälle läßt sich eine sinnvolle Vorbesetzung angeben, die darauf hinausläuft, daß sich der Wert der Variablen nicht verändert.

Eine solche Vorbesetzung ist aber nur für Zustandsvariable, nicht aber für abhängige Variable möglich. Bei der Definition von abhängigen Variablen darf daher der ELSE-Zweig nicht weggelassen werden.

Der Nachweis der korrekten Semantik beim IF-Konstrukt erfordert sehr viel mehr Aufwand.

Die Vollständigkeit verlangt, daß eine Variable in allen Zweigen eines IF-Konstrukts mit einer Definitionsgleichung vertreten sein muß. Anstelle einer Definitionsgleichung kann in einem IF-Zweig wiederum ein IF-Konstrukt stehen, das seinerseits diese Variable in allen Zweigen zu enthalten hat.

Die Eindeutigkeit verlangt, daß eine Variable, die durch ein IF-Konstrukt bereits vollständig definiert ist, nicht an anderer Stelle nochmals definiert wird.

Beispiel:

```
IF     x >  c  LET  y_1 :=  1;
                    IF    z > 0  LET  y_2 :=  x*x;   END
                    ELSE         LET  y_2 :=  x;     END
               END
ELSIF  x < -c  LET  y_1 := -1;
                    IF    z > 0  LET  y_2 :=  -x*x;  END
                    ELSE         LET  y_2 :=  -x;    END
               END
ELSE           LET  y_1 :=  0;
                    y_2 :=  0;
               END
```

Die abhängigen Variablen y_1 und y_2 sind sowohl eindeutig wie vollständig definiert.

Mit den in diesem Abschnitt vorgestellten sprachlichen Mitteln läßt sich bereits das dynamische Verhalten eines diskreten oder kontinuierlichen Modells beschreiben.

Auf die Syntax für Ausdrücke (expression) wurde an dieser Stelle nicht näher eingegangen, weil sich der Leser an anderen höheren Programmiersprachen orientieren kann. Näheres findet sich zudem in Kapitel 5. Auch sei auf [Esch 90] bzw. [Esch 91] verwiesen.

Es fällt auf, daß sich die vorgestellte Syntax nicht von der vieler anderer höherer Programmiersprachen unterscheidet. Dies mag auf den ersten Blick irritierend sein. Wie es sich aber zeigte, ist der Anwender sehr schnell in der Lage, sich auf die veränderte Bedeutung der Konstrukte einzustellen und zu erkennen, daß sie nicht der Ablaufsteuerung, sondern zur Fallunterscheidung bei der Definition von Variablen dienen. Durch die Überprüfung der Vollständigkeit und Eindeutigkeit durch den Compiler wird ihm die Semantik der Sprache sehr schnell bewußt gemacht.

Neben der Definition der Variablen ist selbstverständlich auch eine Typvereinbarung und Initialisierung erforderlich. Dies behandelt der nächste Abschnitt.

### 2.4.2 Deklaration der Modellvariablen

Das hier vorgestellte Sprachkonzept rechnet sich zu den deklarativen Sprachen. Im Gegensatz zu prozeduralen Sprachen wird kein Programmablauf beschrieben, sondern vielmehr das Verhalten von Variablen in Abhängigkeit der anderen Variablen.

Die Formulierung eines Modells (Programms) konzentriert sich demnach ganz auf die Beschreibung der einzelnen Modellvariablen in ihren Eigenschaften und ihrem Verhalten.

Deshalb sprechen wir auch von einer Modellbeschreibungssprache und grenzen uns damit von den herkömmlichen Simulationssprachen ab.

Die Beschreibung einer einzelnen Modellvariablen umfaßt:

- Bezeichner: eindeutiger Name
- Verwendung:
  Zustandsvariable, Sensorvariable, abhängige Variable
- Wertemenge:
  ganzzahlig, reell, logisch, Aufzählungsmenge
- Zeitverlauf:
  konstant, zeitdiskret, zeitkontinuierlich
- ggf. physikalische Maßeinheit
- Initialialwert (für Zustandsvariablen und Sensorvariablen)
- dynamisches Verhalten der Variablen

Das dynamische Verhalten der Variablen wird durch Definitionsgleichungen beschrieben und wurde bereits im vorangegangenen Abschnitt behandelt.

Die Belegung der Modellvariablen im Anfangszeitpunkt ist für Zustandsvariable und ggf. auch für Sensorvariable durch einen Initialwert vorzugeben. Die Anfangswerte der übrigen Variablen können aus diesen Vorgaben berechnet werden.

Die übrigen Teile der Beschreibung umfassen das, was man herkömmlich im Deklarationsteil eines Programms findet: die Vereinbarung eines Bezeichners und die Festlegung der Eigenschaften.

Da zur Definition des dynamischen Verhaltens einer Variablen auf andere Modellvariablen zurückgegriffen werden muß, und bei der Angabe der Eigenschaften von Variablen auch durch den Benutzer definierte Eigenschaften angegeben werden können, bietet es sich an, die Beschreibung der Variablen auf drei Teile zu verteilen:

1) Einführung von Eigenschaften durch den Benutzer
2) Bezeichner und Eigenschaften der Modellvariablen
3) Dynamisches Verhalten der Modellvariablen

Da die Sprache einem bestimmten Zweck dient, der Modellbeschreibung, lassen sich für die Modellvariablen neben der Wertemenge eine Anzahl weiterer Eigenschaften angeben. Diese Eigenschaften ermöglichen eine eingehende Prüfung der Dynamikbeschreibung auf semantische Korrektheit.

a) Verwendung
Der Verlauf einer Zustandsvariablen darf nur durch eine Zustandsüberführungsfunktion definiert sein, bei reeller Modellzeit demnach durch ein Ereignis oder eine Differentialgleichung. Der Verlauf einer abhängigen Variablen muß durch eine algebraische Gleichung definiert werden. Der Verlauf von Sensorvariablen darf - für den Fall, daß sie nicht mit einer anderen Variablen verbunden ist - durch eine algebraische Gleichung definiert werden, die aber als einzige Variable lediglich die Zeit enthalten darf.

b) Wertemenge
Bei der Bildung von Ausdrücken stellen Funktionen und Operatoren Bedingungen an die Wertemengen der Operanden. Bei Wertzuweisungen darf einer Variablen darf nur ein Ausdruck zugeordnet werden, der die gleiche Wertemenge besitzt wie die Variable. Anstelle der geforderten Wertemenge ist auch eine Teilmenge zulässig, wie z.B. eine INTEGER-Zahle anstelle einer REAL-Zahl.

c) Zeitverlauf
Im Dynamikteil darf einer Konstanten kein Wert zugewiesen werden.

Das Verhalten einer diskreten Zustandsvariable darf nur durch eine Ereignisfunktion beschrieben werden. Das Verhalten einer kontinuierlichen Zustandsvariable darf auch durch eine Differentialgleichung beschrieben werden.

Einer diskreten abhängigen Variablen oder Sensorvariablen darf nur ein Ausdruck mit zeitdiskretem Verlauf zugeordnet werden.

d) physikalische Maßeinheit
Zwei Modellgrößen dürfen nur dann summiert werden, wenn ihre Maßeinheiten identisch sind. Zwei Modellgrößen dürfen nur dann verglichen werden, wenn ihre Maßeinheiten kommensurabel, d.h. ineinander überführbar, sind. Bei der Verwendung von Standardfunktionen lassen sich ähnliche Bedingungen abprüfen.

Ein Ausdruck darf einer Modellgröße nur zugeordnet werden, wenn ihre Einheiten kommensurabel sind.

Auf die Angabe einer Syntax für das Festlegen der Eigenschaften wird vorläufig verzichtet, da dies für das Verständnis der Beispiele nicht erforderlich ist und im Kapitel 5 nachgeholt wird. Die Notation ist weitestgehend selbsterklärend und daher mögen an dieser Stelle einige Beispiele genügen.

Beispiel:

```
DEFINITION
    VALUE SET  Color ( 'red', 'green', blue')

DECLARATIONS
    STATE VARIABLE
       CONSTANT     a  (INT)     :=  1
       DISCRETE     Z1 (Color)   :=  'red'
       CONTINUOUS   Z2 (REAL)    := 10

    DEPENDENT VARIABLE
       CONSTANT     F  (Color)
       DISCRETE     Y1 (LOGICAL)
       CONTINUOUS   Y2 (REAL)

    SENSOR VARIABLE
       CONSTANT     m (INT)          :=   0
       DISCRETE     x (REAL [m])     :=  10 [m]
       CONTINUOUS   v (REAL [m/s])   :=   0 [m/s]
```

Die Möglichkeit, Maßeinheiten definieren zu können, ist zwar für eine sichere Modellerstellung von großer Bedeutung, für die Zielsetzung dieser Arbeit sind Maßeinheiten jedoch ohne Belang. Sie werden in Kapitel 6 in aller Kürze abgehandelt.

### 2.4.3 Anfangsbelegung der Modellvariablen

Die Anfangsbelegung der Modellvariablen ist mit der Initialisierung der Zustandsvariablen vollständig bestimmt. Damit eine Vorbesetzung in jedem Fall sichergestellt ist, sind Zustandsvariablen bei ihrer Deklaration in der Modellbeschreibung mit einem Anfangswert zu initialisieren. In höheren Komponenten (siehe Kapitel 2.6) kann diese Vorbesetzung überschrieben werden. Letztendlich gültig ist aber der Initialwert, den der Experimentator über die Programmschnittstelle (siehe Kapitel 1.2) vorgibt, wenn dieser mit einer veränderten Vorbesetzung arbeiten möchte.

Die Werte von nicht angeschlossenen Sensorvariablen im Anfangszeitpunkt ermitteln sich aus einer anzugebenden Zeitfunktion. Da nicht angeschlossene Sensorvariablen in den meisten Fällen mit einem konstanten Wert belegt werden, bietet es sich an, für Sensorvariablen ebenso wie für Zustandsvariablen eine Initialisierung bei ihrer Deklaration zu verlangen. Die Angabe der Zeitfunktion kann dann optional erfolgen.

Die Anfangswerte der angeschlossenen Sensorvariablen sind durch die Verbindung mit einer Zustandsvariablen oder abhängigen Variablen festgelegt.

Die Werte der abhängigen Variablen im Anfangszeitpunkt lassen sich durch die zugehörigen Definitionsgleichungen vollständig aus den Initialwerten der Zustandsvariablen, der Sensorvariablen sowie anderer abhängiger Variablen berechnen.

### 2.4.4 Aufbau einer Basiskomponente

Entsprechend der vorgestellten Aufteilung wird ein Modell bzw. (im Vorgriff auf später) eine Modellkomponente strukturiert.

```
basic_component ::=  BASIC COMPONENT  identifier

                     DEFINITION[S]
                     value_set_definitions
                     unit_definitions
                     function_definitions

                     DECLARATION[S]
                     variable_declaration_part

                     DYNAMIC BEHAVIOUR
                     statement_sequence

                     END OF  identifier
```

Der Abschnitt mit Definitionen ist optional und dient der Vereinbarung von Aufzählungsmengen, benutzereigenen Maßeinheiten sowie von tabellarischen und algorithmischen Funktionen (siehe Kapitel 6).

## 2.5 Algorithmus für die Simulation einer expliziten Modelldarstellung

### 2.5.1 Der Grundalgorithmus

Der folgende Grundalgorithmus simuliert ein formales Modell, das in einer expliziten Modelldarstellung abgefaßt und in einer Komponente (d.h. nicht modular) aufgebaut ist:

```
1)  Anfangszeit und Anfangszustand setzen
    -------------------------------------
    k = 0,  t_k = t_0
    Z(t_0) = Z_0;

2)  Alle (nicht angeschlossenen) Sensorvariablen x_i(t_k) berechnen
    ---------------------------------------------------------------
    Fuer alle i aus {1..NX}:
    x_i(t_k) =  e_i (t_k);

3)  Alle abhaengigen Variablen y_i(t_k) berechnen
    ---------------------------------------------
    Fuer alle i von 1 bis NY:
    y_i(t_k) =  g_i (X(t_k), y_1(t_k), ... , y_i-1(t_k), Z(t_k), t_k);

4)  Alle neuen Zustandsvariablen z_i(t_k+1) berechnen
    -------------------------------------------------
    Fuer alle i aus {1..NZ}:
    z_i(t_k+1) =  f_i (X(t_k), Y (t_k), Z(t_k), t_k);

5)  Variablen des aktuellen Zustands aufzeichnen
    --------------------------------------------
    Monitor ();

6)  Zeitfortschaltung t_k --> t_k+1
    -------------------------------
    k <-- k+1;

7)  Fahre fort mit Schritt 2
```

Dieser Algorithmus setzt voraus, daß die statischen Beziehungen sortiert sind, d.h. stets nur auf bereits berechnete abhängige Variablen zurückgegriffen wird. Die dynamischen Beziehungen und die Belegungen der Sensorvariablen können in jeder beliebigen Reihenfolge berechnet werden.

Die eigentlichen Modellgleichungen befinden sich in den Abschnitten 2-4. Nach Bestimmung der abhängigen Variablen ist der Zustand zum Zeitpunkt $t_k$ vollständig ermittelt. An dieser Stelle, aber auch noch bevor die Zeit weitergesetzt wird, können die interessierenden Variablen aufgezeichnet werden (Monitor).

Gegenüber dieser eher mathematischen Darstellung des Algorithmus ist es bei einer Implementierung in einer prozeduralen Sprache natürlich nicht erforderlich, daß für jede Variable der gesamte Zeitverlauf mitgeführt wird. Es ist lediglich notwendig, daß der Zustand in zwei getrennten Zustandsvektoren gespeichert wird, nämlich einen für den aktuellen Zustand zum Zeitpunkt $t_k$ sowie einen für den Folgezustand zum Zeitpunkt $t_{k+1}$.

Daten:

| | |
|---|---|
| `t` | aktueller Zeitpunkt $t_k$ |
| `t^` | naechster Zeitpunkt $t_{k+1}$ |
| `Z  [NZ]` | `NZ` Zustandsvariablen (Wert zum Zeitpunkt $t_k$) |
| `Z^ [NZ]` | `NZ` Zustandsvariablen (Wert zum Zeitpunkt $t_{k+1}$) |
| `Y  [NY]` | `NY` abhaengige Variablen (Wert zum Zeitpunkt $t_k$) |
| `X  [NX]` | `NX` Sensorvariablen (Wert zum Zeitpunkt $t_k$) |

Algorithmus:

```
1)  Anfangszeit und Anfangszustand setzen
    -------------------------------------
    t = t_0;
    Z = Z_0;

2)  Alle (nicht angeschlossenen) Sensorvariablen x_i berechnen
    ----------------------------------------------------------
    Fuer alle i aus {1..NX}:
    x_i =  e_i (t);

3)  Alle abhaengigen Variablen y_i berechnen
    ----------------------------------------
    Fuer alle i von 1 bis NY:
    y_i =  g_i (X, y_1, ... , y_i-1, Z, t);

4)  Alle neuen Zustandsvariablen z_i^ fuer den Zeitpunkt t^ berechnen
    -----------------------------------------------------------------
    Fuer alle i aus {1..NZ}:
    z_i^ =  f_i (X, Y, Z, t);

5)  Variablen des aktuellen Zustands aufzeichnen
    --------------------------------------------
    Monitor ();

6a) Neuen Zeitpunkt t^ bestimmen und Zeit weiterschalten
    ----------------------------------------------------
    t^ =  ... ;
    t  =  t^;

6b) Alle Zustandsvariablen umspeichern
    ----------------------------------
    Fuer alle i aus {1..NZ}:
    z_i =  z_i^;

7)  Fahre fort mit Schritt 2
```

Bei Verwendung einer eindimensionalen, diskreten Zeitmenge ergibt sich der nächste Zeitpunkt einfach als nächster Wert aus dieser Menge (z.B.: $\hat{t} = t + \Delta t$ ).

Läßt man eine kontinuierliche Zeitmenge mit reellen Zahlen zu und verwendet man als Implementierungszeit eine zweidimensionale Zeitmenge, die sich aus Zeitpunkt und Moment zusammensetzt, dann hängt der Zeitfortschritt vom aktuellen Zustand ab. Solange der aktuelle Zustand ein weiteres Ereignis auslöst, wird mit einem infinitesimal kleinen Zeitinkrement, d.h. mit dem nächsten Moment fortgefahren. Ansonsten wird die Zeit entsprechend dem Auflösungsvermögen um einen endlichen Zeitschritt weitergesetzt. Eine effiziente Implementierung, die nicht für jeden Zeitpunkt einen Schleifendurchlauf benötigt, wurde bereits in [Esch 90] angegeben.

Dort wurde auch beschrieben, wie der Algorithmus zu erweitern ist, wenn neben Ereignissen Differentialgleichungen als Zustandsübergangsfunktionen zulässig sind. Da dies auf die folgenden Überlegungen keinen Einfluß hat, nehmen wir stets auf den angegebenen Grundalgorithmus Bezug, der sich für alle Arten von diskreten Zustandsübergängen eignet.

### 2.5.2 Sortieren der statischen Beziehungen

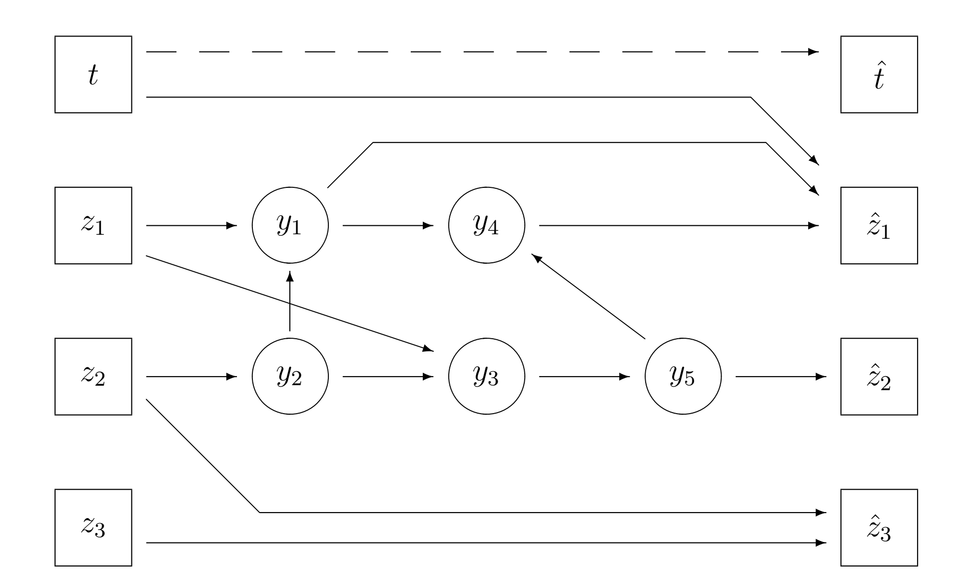
Anhand des eingeführten Abhängigkeitsgraphen kann man sowohl eine geeignete Reihenfolge für die Berechnung der abhängigen Variablen als auch eine insgesamt effizientere Abarbeitung des Algorithmus ableiten.

Wir betrachten das folgende Beispiel mit den angegebenen Abhängigkeiten.

Beispiel: Gegeben sind die funktionalen Abhängigkeiten

$$
\begin{array}{lll lll}
y_1 & := & g_1(z_1,\ y_2); & \hat{z}_1 & := & f_1(y_1,\ y_4,\ t); \\
y_2 & := & g_2(z_2); & \hat{z}_2 & := & f_2(y_5); \\
y_3 & := & g_3(z_1,\ y_2); & \hat{z}_3 & := & f_3(z_2,\ z_3); \\
y_4 & := & g_4(y_1,\ y_5); & & & \\
y_5 & := & g_5(y_3); & & &
\end{array}
$$

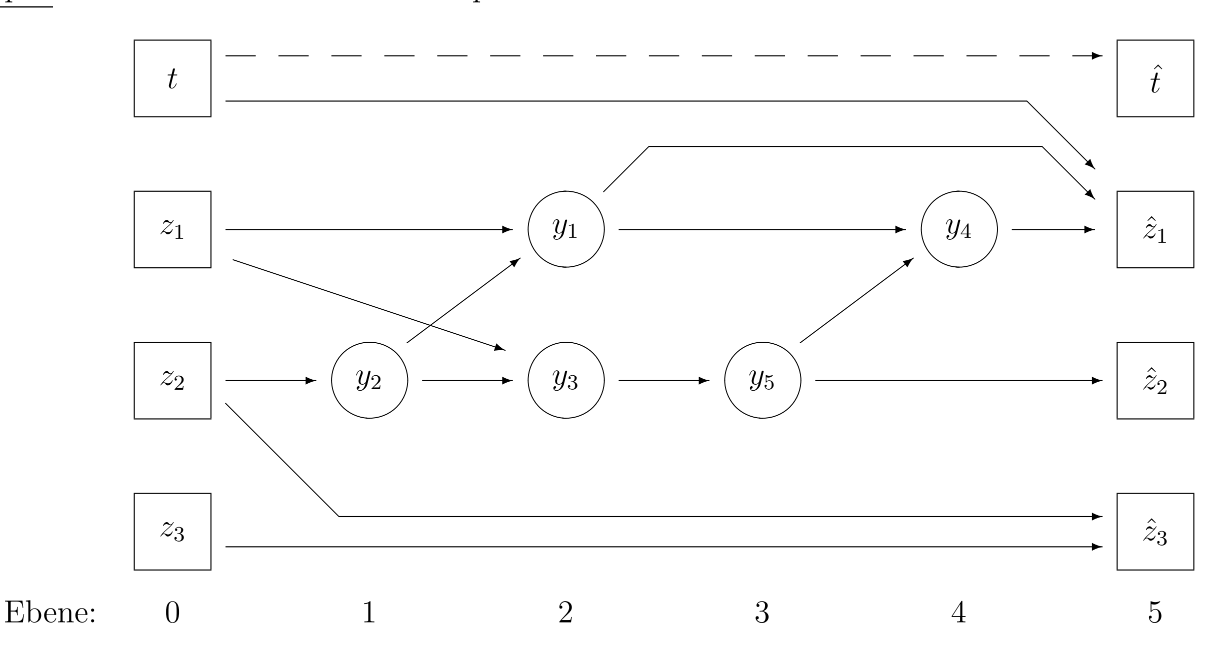
Der dazugehörige Abhängigkeitsgraph hat folgendes Aussehen:



Wir formen diesen Graphen nun so um, daß seine Pfeile nur noch nach vorne orientiert sind und sprechen dann von einem vorwärtsorientierten Graphen. Diese Umformung ist natürlich nur möglich, wenn der Graph zyklenfrei, d.h. die Modellbeschreibung semantisch korrekt ist.



Beispiel: Vorwärtsorientierter Graph




Die Modellvariablen sind in diesem Graphen ebenenweise angeordnet. Alle Variablen einer Ebene beruhen auf den Variablen der davorliegenden Ebenen. Die Zustandsvariablen werden (der Übersichtlichkeit halber) alle in die letzten Ebene gelegt. Jeder Knoten und damit jede Modellvariable erhält die Nummer der Ebene im vorwärtsorientierten Graphen zugeordnet. Anhand dieser Nummer lassen sich die Variablen nun sortieren. Variablen in der gleichen Ebene können in jeder beliebigen Reihenfolge, also auch parallel, berechnet werden.

Beispiel: Der Graph besitzt 5 Ebenen. Die Reihenfolge nach der Sortierung lautet:
$y_2 \ , \ y_1 \ || \ y_3 \ , \ y_5 \ , \ y_4 \ \ ; \ \ \hat{z}_1 \ || \ \hat{z}_2 \ || \ \hat{z}_3$

Vom Gesichtspunkt der Effizienz ist der Algorithmus aber noch keineswegs optimal. Es werden nämlich zu jedem neuen Zeitpunkt sämtliche Modellvariablen neu bestimmt.

### 2.5.3 Verbesserung der Effizienz

Speziell in diskreten Modellen verändern nur wenige Zustandsvariablen zu einem Zeitpunkt ihren Wert. Es ist dann nicht erforderlich, den gesamten Graphen zu durchlaufen. Vielmehr ist es ausreichend, nur diejenigen Variablen neu zu berechnen, welche von einer Zustandsvariablen abhängen, die sich verändert hat.

Beispiele: Nach einer Veränderung von $z_3$ ist lediglich $\hat{z}_3$ neu zu bestimmen. Nach einer Veränderung von $z_1$ sind $y_1$, $y_3$, $y_4$, $y_5$ sowie $\hat{z}_1$ und $\hat{z}_2$ neu zu bestimmen. Nach einer Zeitfortschaltung ist $\hat{z}_1$ neu zu bestimmen.

Um zu erreichen, daß nur bestimmte Variablen während eines Durchlaufs neu berechnet werden, setzt man (dynamische) Marken. Hierzu lassen sich zwei Vorgehensweisen angeben:

1) Markierung der Auswirkungen einer Änderung (Vorwärts-Verfolgung): Nach der Veränderung einer Variablen werden diejenigen Variablen markiert, auf welche diese Änderung Einfluß hat. Nur Variablen mit einer Markierung werden neu berechnet.

2) Markierung der veränderten Variablen (Rückwärts-Verfolgung): Nach der Veränderung einer Variablen erhält diese eine Marke. Der Wert einer Variablen wird nur dann neu bestimmt, wenn eine der Variablen, von der diese abhängt, markiert ist.

Auf eine geeignete Vorbesetzung der Marken sowie auf das Rücksetzen an passender Stelle ist zu achten. Die Sortierung der statischen Beziehungen ist nach wie vor erforderlich.

Beispiel: Vorwärts-Verfolgung

Der Algorithmus mit Marken zur Vorwärts-Verfolgung sieht für obiges Beispiel ausschnittsweise etwa so aus:

```
1) Vorbesetzung:  z_i. Flag = FALSE;  fuer alle i aus {1..NZ}
                  y_i. Flag = TRUE;   fuer alle i aus {1..NY}
                  z^_i.Flag = TRUE;   fuer alle i aus {1..NZ}

3) Falls  y_2.Flag  dann  y_2. Value =  g_2 (z_2);
                          y_2. Flag  =  FALSE;
                          Bei Wertaenderung
                          dann
                                y_1. Flag  =  TRUE;
                                y_3. Flag  =  TRUE;
                          ende
                    ende

   Falls  y_1.Flag  dann  y_1. Value =  g_1 (z_1, y_2);
                          y_1. Flag  =  FALSE;
                          Bei Wertaenderung
                          dann
                                y_4. Flag  =  TRUE;
                                z_1^.Flag  =  TRUE;
                          ende
                    ende
   ...
   ...

4) Falls  z_1^.Flag  dann  z_1^.Value =  f_1 (y_1, y_4, t);
                           z_1^.Flag  =  FALSE;
                           Bei Wertaenderung
                           dann
                                 z_1. Flag  =  TRUE;
                           ende
                     ende
   ...
```

```
6a) t^ =  ...;
    t  =  t^;
    Falls realer Zeitfortschritt dann  z_1^.Flag = TRUE;
                                 ende

6b) Falls  z_1.Flag  dann  z_1 = z_1^;
                           z_1. Flag = FALSE;
                           y_1. Flag = TRUE;
                           y_3. Flag = TRUE;
                     ende
    Falls  z_2.Flag  dann  z_2 = z_2^;
                           z_2. Flag = FALSE;
                           y_2. Flag = TRUE;
                           z_3^.Flag = TRUE;
                     ende
    Falls  z_3.Flag  dann  z_3 = z_3^;
                           z_3. Flag = FALSE;
                           z_3^.Flag = TRUE;
                     ende
```

Die Ziffern geben an, zu welchem Teil des Grundalgorithmus der jeweilige Ausschnitt gehört.

Bei der Rückwärts-Verfolgung werden die Marken aller Variablen abgefagt, welche auf die zu berechnende Variable Einfluß haben.

Beispiel: Rückwärts-Verfolgung

Der Algorithmus mit Marken zur Rückwärts-Verfolgung sieht für obiges Beispiel ausschnittsweise etwa so aus:

```
 1) Vorbesetzung:    t. Flag = TRUE;
                   y_i. Flag = TRUE;   fuer alle i aus {1..NY}
                   z_i. Flag = TRUE;   fuer alle i aus {1..NZ}
                   z_i^.Flag = FALSE;  fuer alle i aus {1..NZ}

 3) Falls  z_2.Flag  dann  y_2. Value =  g_2 (z_2);
                           Bei Wertaenderung
                           dann
                                 y_2. Flag  =  TRUE;
                           ende
                     ende
    Falls  z_1.Flag
     oder  y_2.Flag  dann  y_1. Value =  g_1 (z_1, y_2);
                           Bei Wertaenderung
                           dann
                                 y_1. Flag  =  TRUE;
                           ende
                     ende
    ...
    ...
```

```
4) Falls    t.Flag
    oder  y_1.Flag
    oder  y_4.Flag  dann  z_1^.Value =  f_1 (y_1, y_4, t);
                          Bei Wertaenderung
                          dann
                                z_1^.Flag  =  TRUE;
                          ende
                    ende

   Falls  y_5.Flag  dann  z_2^.Value =  f_2 (y_5);
                          Bei Wertaenderung
                          dann
                                z_2^.Flag  =  TRUE;
                          ende
                    ende

   Falls  z_2.Flag
    oder  z_3.Flag  dann  z_3^.Value =  f_3 (z_2, z_3);
                          Bei Wertaenderung
                          dann
                                z_3^.Flag  =  TRUE;
                          ende
                    ende

6a) t^ =  ...;
    t  =  t^;
    Falls realer Zeitfortschritt  dann  t.Flag = TRUE;
                                  sonst t.Flag = FALSE;

6b) Fuer alle i aus {1..NY}:
    y_i.Flag = FALSE;

    Fuer alle i aus {1..NZ}:
    z_i.Flag = FALSE;

    Fuer alle i aus {1..NZ}:
    Falls  z_i^.Flag dann  z_i = z_i^;
                           z_i. Flag = TRUE;
                           z_i^.Flag = FALSE;
                     ende
```

Für eine manuelle Programmierung ist diese Lösung wohl eher geeignet, da sie einen einfachen schematischen Aufbau besitzt.

Von besonderer Bedeutung für das Übersetzen der Modellbeschreibung in einen ausführbaren Code hat sich das Aufstellen des Abhängigkeitsgraphen erwiesen. Basierend auf dem Abhängigkeitsgraphen wird die Zyklenfreiheit der Modellbeschreibung überprüft, die algebraischen Gleichungen geordnet und durch das Einbringen dynamischer Marken optimaler Code erzeugt.

Es empfiehlt sich, auch die zeitabhängigen Sensorvariablen mit in den Abhängigkeitsgraph aufzunehmen und genauso wie die abhängigen Größen zu behandeln. Der Übersichtlichkeit

halber wurde im Beispiel jedoch auf diese Größen verzichtet.

Die Ausführungen haben gezeigt, daß es möglich ist, das vorgestellte Programmiermodell effizient zu bearbeiten. Der dazu erforderliche Code läßt sich ohne Schwierigkeit von einem Compiler erzeugen. Die Anzahl der zur Laufzeit ausgeführten Operationen ist minimal und kann durch manuelle Programmierung nicht weiter reduziert werden.

Allerdings hängt der Effekt, den die eben geschilderten Verbesserungen bringen, sehr stark davon ab, wie viele Zustandsvariablen zu einem Zeitpunkt ihren Wert verändern.

In einem kontinuierlichen Modell oder einem zeitlich getakteten Modell ändern sich in der Regel mit jedem Zeitpunkt sämtliche Zustandsvariablen. Hier wird die Einführung von Marken natürlich gar keine Wirkung zeigen. Nur bei geschalteten, kontinuierlichen Vorgängen läßt sich eine Verbesserung der Effizienz erwarten.

In einem diskreten Modell allerdings ändern sich zu einem Zeitpunkt stets nur sehr wenige Zustandsvariablen. Dementsprechend müssen auch nur wenige Größen weiterverfolgt werden. Das vorgestellte Verfahren erweist sich hier um so wirkungsvoller, je größer das gesamte Modell ist.

Der Algorithmus erscheint so effizient, daß vermutet werden kann, daß sich auch durch Parallelisierung keine nennenswerte Verkürzung der Rechenzeiten erreichen läßt. Die bisher allesamt erfolglosen Versuche, diskrete Modelle auf Parallelrechnern zu simulieren, geben dieser Vermutung recht.

Im Kapitel über Modularisierung wird untersucht, ob und inwieweit sich diese effiziente Abarbeitung bei einer komponentenweisen Zerlegung des Modells aufrechterhalten läßt. Dort werden auch die beiden vorgestellten Verfahren zur Markierung nochmals diskutiert werden.

## 2.6 Modularisierung

### 2.6.1 Anforderungen

Die Modularisierung soll – ganz allgemein gesehen – ein Programm in abgegrenzte Einheiten überschaubarer Größe zerlegen, um es

- besser handhabbar zu machen
- gleichartiges Verhalten nicht mehrfach zu beschreiben.

Für ein Modularisierungskonzept, d.h. ein Konzept, das die Zerlegung eines Modells in modulare Bestandteile, sprich Komponenten gestattet, gelten die folgenden Anforderugen:

1) Ein modularisiertes Modell muß exakt das gleiche Verhalten zeigen wie ein Modell, das nicht zerlegt ist.
2) Die Prüfbarkeit der Modellsemantik auf Korrektheit (Eindeutigkeit etc.) muß ebenso gewährleistet sein.
3) Es darf zu keinen nennenswerten Einbußen an Laufzeiteffizienz kommen.
4) Dem Anwender dürfen keine Restriktionen auferlegt werden, nach welchen Gesichtspunkten er die Modularisierung vorzunehmen hat.
5) Eine Hierarchiebildung muß möglich sein.
6) Die Formulierung einer Komponente muß unabhängig von der Umgebung sein, in der sie später zum Einsatz kommt.
7) Jede Modellkomponente sollte auch wie ein selbständiges Modell betrieben werden können, um ein Modell nach und nach aufbauen und testen zu können.
8) Die Komponenten sollten autonom sein, d.h. jede Komponente verändert nur ihre eigenen Variablen, beeinflußt aber nicht direkt die Variablen fremder Komponenten.

Für eine explizite Modelldarstellung bietet sich ein sehr einfaches Konzept an, das die genannten Anforderungen erfüllt.

### 2.6.2 Modellzerlegung und Kommunikation zwischen Komponenten

Die Attribute (Konstanten und Variablen) des Modells und deren Definitionsgleichungen werden auf einzelne Komponenten verteilt. Diejenigen Variablen, welche in den Modellgleichungen benötigt werden, aber nicht in der Komponente enthalten sind, werden durch Sensorvariablen ersetzt.

Beispiel: Ein Modell mit

```
z^  := f (y_1, y_2, z);
y_1 := g (z);
y_2 := g (y_1, z);
```

soll in zwei Komponenten zerlegt werden. Eine Komponente soll die Variablen `z` und `y_2` enthalten, die andere `y_1`.

```
Komponente A:                   Komponente B:
z^  := f (x_A, y_2, z);         y_1 := g (x_B);
y_2 := g (x_A, z);
```

Die Sensorvariablen `x_A` und `x_B` wurden hierzu neu eingeführt.

Durch diese Art der Modularierung werden Modelle geschaffen, bei denen der Verlauf der Eingangsfunktionen nicht näher bestimmt ist.

Ein Gesamtmodell entsteht, wenn die beiden Komponenten im Verbund betrieben werden. Hierzu liefern die abhängigen Variablen und Zustandsvariablen der einen Komponente den Zeitverlauf für die Sensorvariablen der anderen Komponente.

Die Modellbeschreibung ist daher zu ergänzen und den Sensorvariablen jeweils eine interne Variable einer anderen Komponente zuzuordnen. Unter internen Variablen verstehen wir Zustandsvariablen und abhängige Variablen, da diese der Komponente selbst zugerechnet werden. Aus der Sicht einer Komponente ist hingegen eine Sensorvariable eine externe Variable.

Beispiel:

```
x_A  :=  y_1;
x_B  :=  z;
```

Da jede Komponente eine abgeschlossene Einheit darstellen soll, ist es natürlich wünschenswert, daß sich die Bezeichner nicht von denen anderer Komponenten unterscheiden müssen. Damit im Gesamtmodell Variablen mit gleichen Namen unterscheidbar bleiben, wird der Komponentennamen hinzugezogen.

Beispiel: Komponente `A`: / Komponente `B`:

```
z^ := f (x, y, z);          y := g (x);
y  := g (x, z);
```

Die Zuordnungen an die Sensorvariablen werden beschrieben durch:

```
A.x  :=  B.y;
B.x  :=  A.z;
```

Eine Komponente ist demnach ein (Teil-) Modell mit eigenen Variablen und eigenen Definitionsgleichungen für diese Variablen.

Bei der Zuordnung eines Wertes an eine Sensorvariable beschränken wir uns auf die Zuordnung einer internen Variablen. Ein Ausdruck, der sich aus internen Variablen zusammensetzt oder ein einzelner Wert dürfen nicht zugeordnet werden. Wir sprechen daher im folgenden nicht mehr von Zuordnung sondern von Verbindung und verwenden einen Pfeil `'-->'` als Verbindungsoperator.

Beispiel:

```
EFFECT CONNECTIONS
A.z  -->  B.x;
B.y  -->  A.x;
```

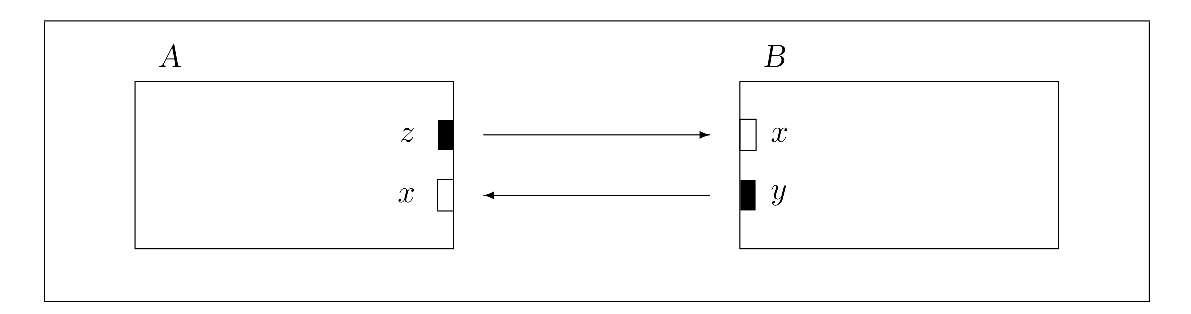
Die internen Variablen auf der linken Seite werden mit den Sensorvariablen auf der rechten Seite verbunden.

Diese Maßnahme fördert die Übersichtlichkeit und ermöglicht eine graphische Darstellung der Modellbeschreibung. Sie ist keine Einschränkung der Modellierbarkeit. Schließlich ist es immer möglich, für Verknüpfungen zwischen mehreren internen Variablen eine weitere Komponente zwischenzuschalten.

Modellbeschreibung spielt sich demnach auf zwei Ebenen ab. Auf der unteren Ebene in den sogenannten Basiskomponenten wird durch Verknüpfung zwischen Variablen das dynamische Verhalten definiert. Auf der übergeordneten Ebene, in den sogenannten höheren Komponenten, wird durch Verbindungen von Sensorvariablen und internen Variablen eine Modellstruktur aufgebaut.

Die Modellstruktur mit Komponenten und Verbindungen läßt sich sehr schön graphisch veranschaulichen.

Beispiel:




Innerhalb eines Komponenten-Kästchens stehen die komponentenspezifischen Bezeichner, am Rahmen des Kästchens der Name der Komponente. Die Pfeile weisen von einer internen Variablen auf eine oder mehrere Sensorvariablen und sind als Wirkpfeile zu verstehen. Sie symbolisieren, daß von einer Komponente eine Wirkung auf eine andere Komponente ausgeht.

Durch eine Verbindung zwischen zwei Komponenten wird einer Komponente Information aus einer anderen Komponente zugänglich gemacht. Es handelt sich demnach um einen lesenden Zugriff.

Schreibende Zugriffe sind in diesem Konzept nicht vorgesehen. Die Veränderung einer Variablen kann nur von der betreffenden Komponente selbst vorgenommen werden. Wir bezeichnen Komponenten deshalb auch als äutonom:'

Die Gültigkeit der Variablen beschränkt sich nicht auf die Basiskomponente, in der sie deklariert werden. Sie sind auch in der darüberliegenden, höheren Komponente bekannt.

Da den Sensorvariablen innerhalb einer Komponente bereits eine Zeitfunktion (oder auch nur ein konstanter Wert) zugeordnet werden kann, ist es möglich, eine Komponente auch dann als vollständiges Modell zu betreiben, wenn einzelne Sensorvariable nicht angeschlossen ist. Dies ist zu Testzwecken wünschenswert, aber auch um ein Modell nach und nach (Bottom-up) aufzubauen.

### 2.6.3 Klassenbildung

Ein Modell läßt sich u.U. so zerlegen, daß gleichartige Komponenten auftreten. Sie besitzen zwar ein identisches dynamisches Verhalten, aber ihre Variablen unterscheiden sich in ihren aktuellen Werten. In diesem Fall genügt natürlich, die Beschreibung dieser Komponenten nur ein einziges Mal niederzulegen. Um solche gleichartigen Komponenten zu unterscheiden, kann man sie entweder mit verschiedenen Indizes oder verschiedenen Namen versehen.

Die Beschreibung einer Komponente vereinbart demnach eine *Klasse* von Komponenten, von der mehrere *Ausprägungen* (instances, incarnations) in einem Modell enthalten sein könnnen.

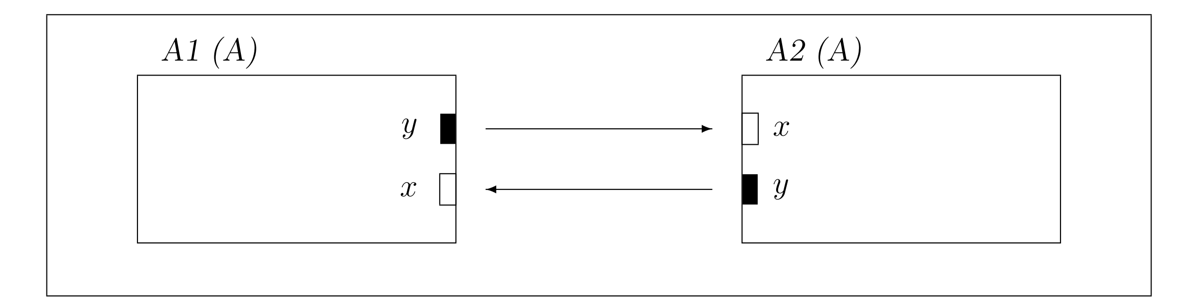
Beispiel: Komponentenklasse `A`:

```
z^ := f (x, y, z);
y  := g (x, z);
```

Modell mit Ausprägungen `A1` und `A2` der Klasse `A`:

```
SUBCOMPONENTS  A1  OF CLASS  A,
               A2  OF CLASS  A

EFFECT CONNECTIONS
A1.y  -->  A2.x;
A2.y  -->  A1.x;
```




Am Rahmen eines Kästchens steht nun in der graphischen Darstellung hinter dem Ausprägungsnamen in Klammern der Klassenname der Komponente.

Eine Belegung des Anfangszustands ist bereits innerhalb der Komponenten vorgesehen. Diese Vorbesetzung hat jedoch für alle Ausprägungen einer Komponentenklasse Gültigkeit. Deshalb ist dafür zu sorgen, daß die Variablen jeder einzelnen Ausprägung vorbelegt werden können.

Beispiel:

```
A1.z := 1;
A2.z := 2;
```

### 2.6.4 Prüfung der semantischen Korrektheit

Ein Modell, das in Komponenten zerlegt ist, muß natürlich den gleichen Anforderungen an die Semantik gerecht werden wie ein nicht zerlegtes Modell. Insbesondere bedeutet dies, daß die Zyklenfreiheit des Abhängigkeitsgraphen nicht verletzt sein darf.

Beispiel:

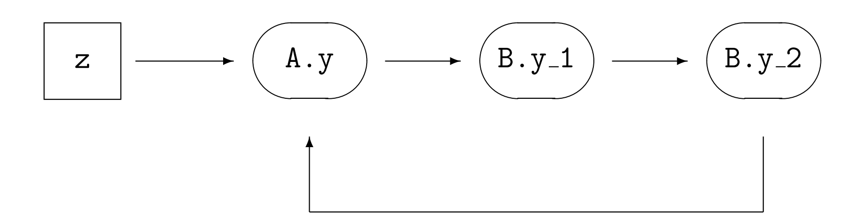
```
Komponente A:                   Komponente B:
z^ := f (x, y, z);              y_1 := g_1 (x);
y  := g (x, z);                 y_2 := g_2 (y_1);
```

Die Verbindungen werden beschrieben durch:

```
A.y    -->  B.x;
B.y_2  -->  A.x;
```

Während die beiden Komponenten für sich genommen semantisch korrekt sind, ergibt sich durch die vorgenommenen Verbindungen ein Zyklus:

```
A.y    :=  g   (B.y_2, z);
B.y_1  :=  g_1 (A.y     );
B.y_2  :=  g_2 (B.y_1   );
```




Um einen solchen Zyklus auf höherer Ebene zu erkennen, müssen nicht die Abhängigkeiten von sämtlichen Modellvariablen bekannt sein. Vielmehr genügt es zu wissen, von welchen Sensorvariablen die Ausgangsvariablen abhängen.

Beispiel: Für die beiden Komponenten `A` und `B` genügt die folgende verkürzte Information:

```
Komponente A:                   Komponente B:
y  := g (x);                    y_1 := g_1 (x);
                                y_2 := g_2 (x);
```

Bei den Verbindungen:

```
A.y    -->  B.x;
B.y_2  -->  A.x;
```

ergibt sich:

```
A.y    :=  g   (B.y_2);
B.y_2  :=  g_2 (A.y  );
```

Wie man sofort sieht, läßt sich auch hieran der Zyklus erkennen.

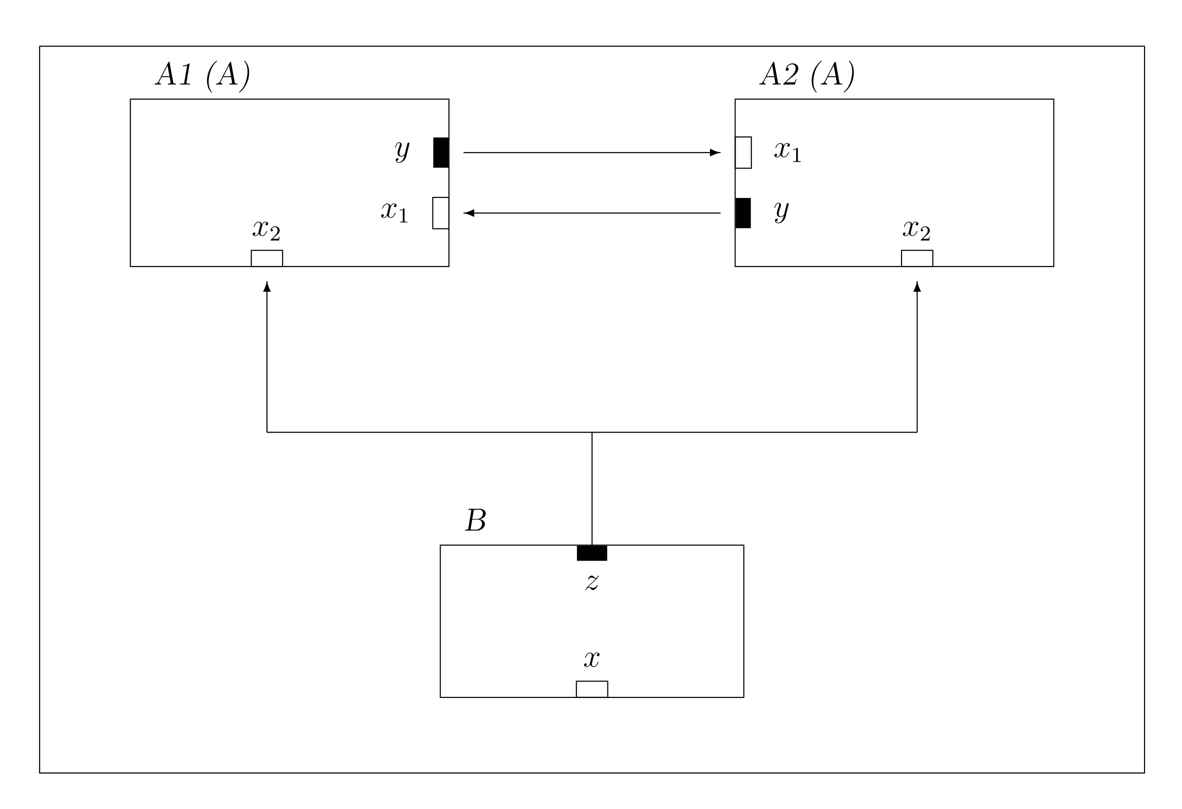
Eine weitere Forderung ist, daß eine Sensorvariable nur mit einer einzigen internen Variablen verbunden sein darf. Dies entspricht der Forderung nach Eindeutigkeit bei der Zuordnung wie sie bereits aus den Basiskomponenten bekannt ist.

Hingegen ist es nicht erforderlich, daß jede Sensorvariable verbunden ist. Bleibt die Schnittstelle offen, dann tritt hierfür die in der Basiskomponente festgelegte Zeitfunktion (oder der feste Wert) ein. Auf die Vollständigkeit bei der Belegung von Sensorvariablen kann daher verzichtet werden.

Dennoch erscheint es ratsam, wenigstens eine Warnung auszugeben, wenn eine Sensorvariable nicht verbunden ist, da hier eine häufige Fehlerquelle liegt.

In der graphischen Darstellung bedeutet dies, daß von einer internen Variablen mehrere Verbindungspfeile ausgehen dürfen, auf eine Sensorvariable jedoch nur ein einziger Verbindungspfeil gerichtet sein darf.

Beispiel:




Die Zustandsvariable `z` der Komponente `B` ist sowohl mit `x_2` der Komponente `A1` wie auch mit `x_2` der Komponente `A2` verbunden. Die Sensorvariable `x` der Komponente `B` bleibt unbelegt.

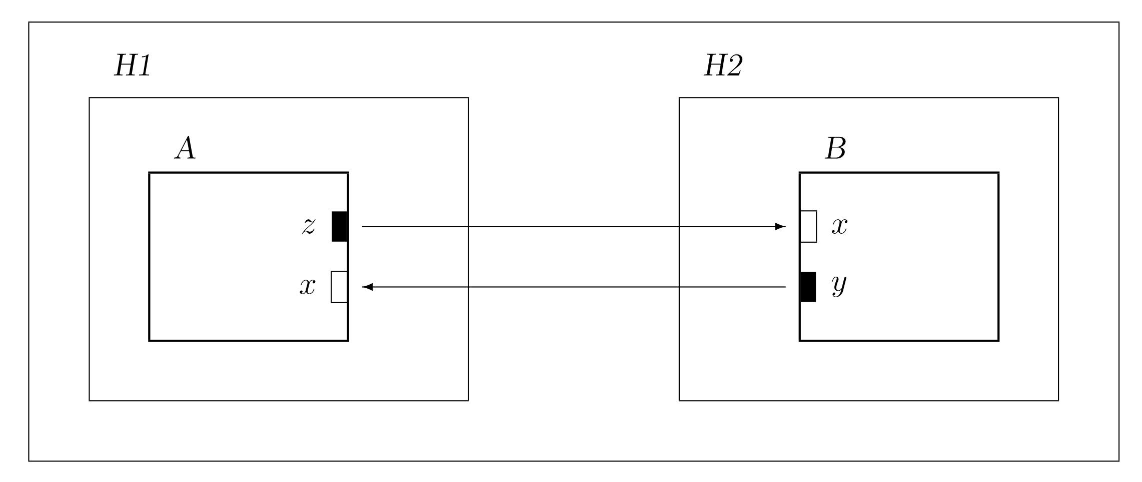
### 2.6.5 Hierarchische Modularisierung

Auch eine Hierarchisierung der Modellzerlegung ist möglich. Zu diesem Zweck faßt man mehrere Komponenten selbst wiederum zu einer Komponente zusammen. Hierdurch entsteht eine Blockstruktur. Die Identifikation der Variablen erfolgt durch einen Pfad, in dem der Reihe nach alle übergeordneten Komponenten enthalten sind.

Beispiel: Variablennamen: `H1.A.z`

Variable `z` aus Komponente `A`, eingebettet in Komponente `H1`.

```
Verbindungen:   H1.A.z  -->  H2.B.x;
                H2.B.y  -->  H1.A.x;
```




Die Beschreibung der Kommunikation über einen mehrstufigen Pfad macht einen tiefen Einblick in die beteiligten Komponenten erforderlich. Es erscheint daher sinnvoll, eine Datenkapselung vorzunehmen und nur bestimmte Variablen einer höheren Komponente für die nächst höhere Schicht sichtbar zu machen.

Zu diesem Zweck ist es erforderlich, Sensorvariablen und interne Variablen für höhere Komponenten einzuführen.

Sensorvariablen höherer Komponenten nehmen Information von außenentgegen und leiten sie an noch nicht angeschlossene Sensorvariablen von untergeordneten Komponenten weiter.

Interne Variablen höherer Komponenten stehen für eine interne Variable einer untergeordneten Komponente und machen deren Information äußenverfügbar.

Beispiel: Eine höhere Komponente `H` enthält die Komponenten `A1`, `A2` und `B` mit Sensorvariablen `x` und internen Variablen `z` bzw. `y`.

```
SUBCOMPONENTS
  A1  OF CLASS  A,
  A2  OF CLASS  A,
  B
```

Sensorvariablen der höheren Komponente: `x_1, x_2`

```
  A1.x  :=  x_1;
  A2.x  :=  x_1;
  B.x   :=  x_2;
```

Interne Variablen der höheren Komponente: `u_11, u_12, u_2`

```
  u_11  :=  A1.z;
  u_12  :=  A2.z;
  u_2   :=  B.y;
```

Da auch hier Variablen einander zugeordnet werden und ein Informationsfluß dargestellt wird, verwenden wir auch in diesem Fall den Verbindungsoperator '-->'. Um weiterhin die Eindeutigkeit sicherzustellen, darf auf Sensorvariable einer Subkomponente nur ein Verbindungsoperator weisen.

Die in höheren Komponenten eingeführten Sensorvariablen und internen Variablen übernehmen alle Eigenschaften der mit ihnen verbundenen Variablen. Eine explizite Deklaration ist daher nicht erforderlich. Beim erstmaligen Auftreten des Bezeichners kann eine Deklaration implizit vorgenommen werden.

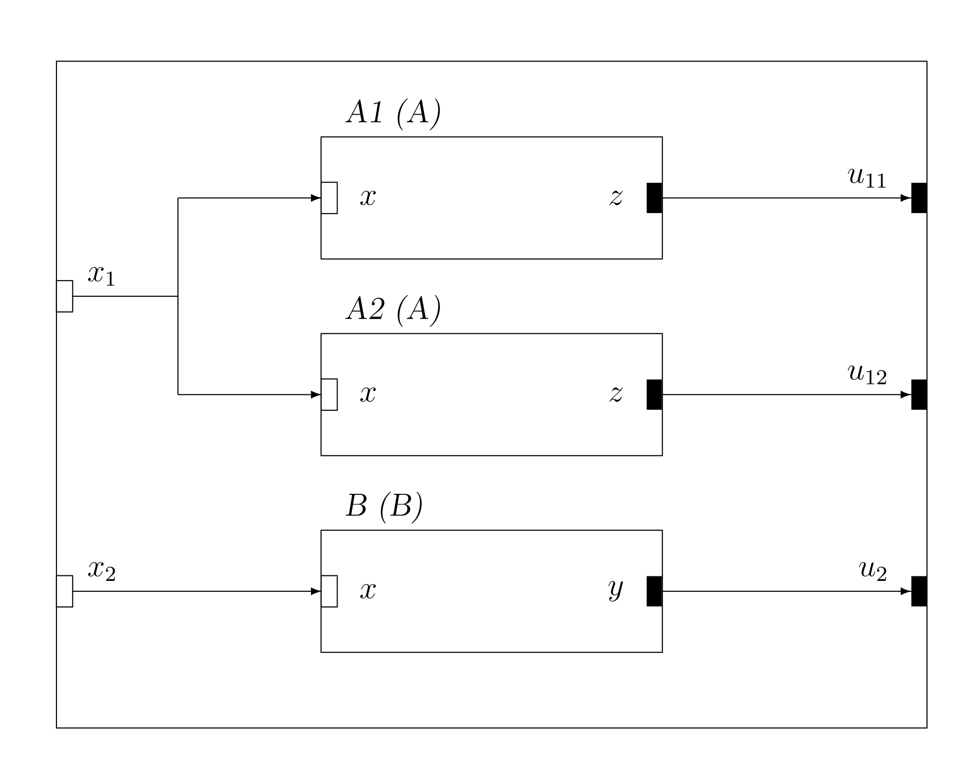
Beispiel:

```
SENSOR VARIABLES
x_1  -->  A1.x;
x_1  -->  A2.x;
x_2  -->  B.x;

INTERNAL VARIABLES
u_11  <--  A1.z;
u_12  <--  A2.z;
u_2   <--  B.y;
```

Auch für hierarchisch strukturierte Modelle gibt es eine eindeutige graphische Darstellung.

Beispiel:



### 2.6.6 Sprachliche Konstrukte für höhere Komponenten

Entsprechend den obigen Ausführungen besteht eine höhere Komponente aus:

1) Deklaration der Subkomponenten
2) Deklaration der Schnittstellenvariablen
   a) Sensor-Variablen
   b) interne Variablen
3) Verbindungen zwischen Komponenten
4) Initialisierung der Modellvariablen in Subkomponenten

Die Syntax der Modellbeschreibungssprache kann sich unmittelbar daran anlehnen und ist im folgenden nochmals zusammengefaßt.

```
high_level_component ::=   HIGH LEVEL COMPONENT  identifier
                           subcomponent_declaration
                         [ sensor_variables ]
                         [ internal_variables ]
                         [ effect_connections ]
                         [ initialization ]
                           END OF  identifier

subcomponent_declaration ::=   SUBCOMPONENT[S]
                               subcomponent  { ',' subcomponent }

subcomponent       ::=   identifier  [ OF CLASS identifier ]

sensor_variables   ::=   SENSOR VARIABLE[S]
                         identifier '-->'  identifier '.' identifier ';'
                       { identifier '-->'  identifier '.' identifier ';' }

internal_variables ::=   INTERNAL VARIABLE[S]
                         identifier '<--'  identifier '.' identifier ';'
                       { identifier '<--'  identifier '.' identifier ';' }

effect_connections ::=   EFFECT CONNECTION[S]
                         identifier '.' identifier  '-->'
                         identifier '.' identifier ';'
                       { identifier '.' identifier  '-->'
                         identifier '.' identifier ';'     }

initialization     ::=   INITIAL CONDITION[S]
                         identifier '.' identifier  ':='  value ';'
                       { identifier '.' identifier  ':='  value ';' }
```

Die Bedeutung der Bezeichner ergibt sich aus den Erläuterungen der vorangegangenen Abschnitte.

## 2.7 Algorithmus für die Simulation einer modularen Modellbeschreibung

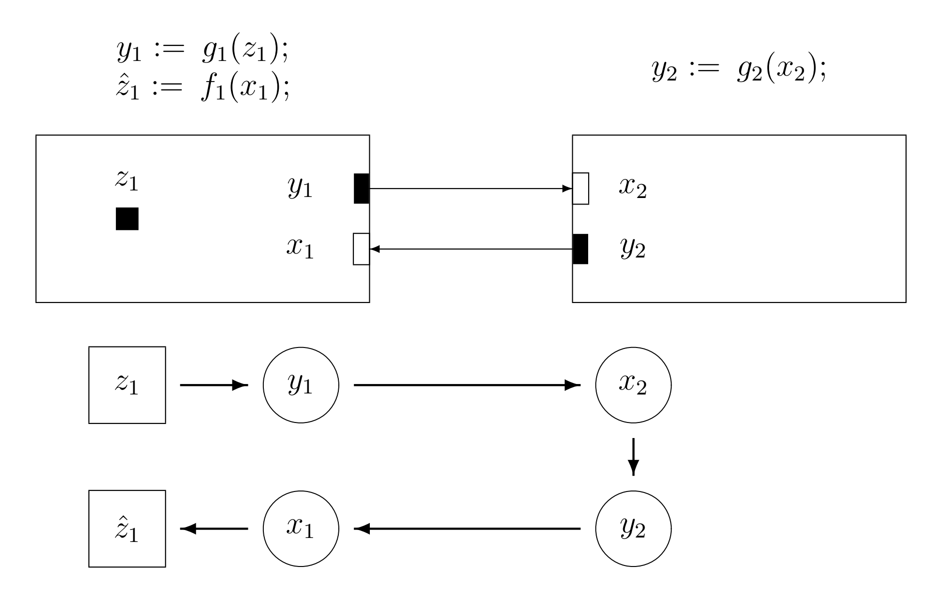
Die Betrachtungen in diesem Abschnitt gehen davon aus, daß man ein modular beschriebenes Modell komponentenweise übersetzen möchte. Wegen des Klassenkonzepts bedeutet dies, daß der übersetzte Code für jede Komponentenklasse nur einmal vorhanden ist, während die Datenbereiche für jede Ausprägung einer Klasse angelegt werden. Neben der Speicherplatzersparnis liegen die Vorteile vor allem darin, daß nach der Änderung einer einzelnen Komponente nicht das gesamte Modell neu übersetzt werden muß.

Verzichtet man auf die getrennte Übersetzbarkeit, lassen sich alle Bestimmungsgleichungen in ein gemeinsames Programm packen. Es kann dann der bereits besprochene Algorithmus eingesetzt werden, wenn zuvor alle Gleichungen des Modells in eine geeignete Reihenfolge gebracht werden.

### 2.7.1 Der Grundalgorithmus für modulare Modelle

Um die getrennte Übersetzbarkeit der Modellkomponenten zu gewährleisten, muß mit einem etwas erweiterten Algorithmus gearbeitet werden. Das liegt daran, daß die Gleichungen auf mehrere Unterprogramme verteilt sind und ein Sortieren der Gleichungen nur noch innerhalb einzelner Modellkomponenten möglich ist.

Beispiel:




In welcher Reihenfolge man auch immer die beiden Komponenten aufruft, eine der beiden Komponenten muß zweimal betreten werden, um $\hat{z}_1$ aus $z_1$ bestimmen zu können.

Die einfachste Art, diesem Problem zu begegnen, ist, die Gleichungen in allen Komponenten sooft auszuwerten wie der Abhängigkeitsgraph Ebenen besitzt. Am Ende des ersten Durchlaufs sind mit Sicherheit die Variablen der Ebene 1 korrekt berechnet, am Ende des zweiten Durchlaufs die der Ebene 2 u.s.w. .

Wir modifizieren daher unseren Grundalgorithmus dahingehend:

Daten:

| | |
|---|---|
| `t` | aktueller Zeitpunkt $t_k$ |
| `t^` | nächster Zeitpunkt $t_k + 1$ |
| | |
| `NK` | Anzahl der Komponenten |
| `NZ [NK]` | Anzahl der Zustandsvariablen pro Komponente |
| `NY [NK]` | Anzahl der abhängigen Variablen pro Komponente |
| `NX [NK]` | Anzahl der Eingangsvariablen pro Komponente |
| `NE` | Anzahl der Ebenen des Abhängigkeitsgraphen |

Fuer jede Komponente $j$:

| | |
|---|---|
| `Z  [NZ_j]` | `NZ_j` Zustandsvariablen (Wert zum Zeitpunkt $t_k$) |
| `Z^ [NZ_j]` | `NZ_j` Zustandsvariablen (Wert zum Zeitpunkt $t_{k+1}$) |
| `Y  [NX_j]` | `NX_j` Ausgangsvariablen (Wert zum Zeitpunkt $t_k$) |
| `X  [NX_j]` | `NX_j` Eingangsvariablen (Wert zum Zeitpunkt $t_k$) |

Algorithmus:

```
1)  Anfangszeit und Anfangszustand setzen
    -------------------------------------
    t = t_0
    Z = Z_0;

    Schleife ueber alle Zeitpunkte
    ==============================
    {
       Schleife ueber alle Ebenen
       ==========================
       Fuer alle m von 1 .. NE:
       {
          Schleife ueber alle Komponenten
          ===============================
          Fuer alle j aus {1..NK}:
          {
             Aufruf der Komponente j
             =======================

2)           Alle nicht verbundenen Sensorvariablen x_i berechnen
             ----------------------------------------------------
             Falls Ebene  m = 1:
               Fuer alle nicht verbundenen x_(i,j) mit i aus {1..NX_j}:
               x_(i,j) =  e_(i,j) (t);

3)           Alle abhaengigen Variablen y_i berechnen
             ----------------------------------------
             Falls Ebene  m < NE:
               Fuer alle i aus {1..NY_j}:
               y_(i,j) =  g_(i,j) (X_j, Y_j, Z_j, t);
```

```
4)          Alle neuen Zustandsvariablen z_i^ berechnen
            -------------------------------------------
            Falls Ebene  m = NE:
              Fuer alle i aus {1..NZ_j}:
              z_(i,j)^ =  f_(i,j) (X_j, Y_j, Z_j, t);
         }
         Ende Schleife ueber alle Komponenten
         ====================================
      }
      Ende Schleife ueber alle Ebenen
      ===============================

5)    Variablen des aktuellen Zustands aufzeichnen
      --------------------------------------------
      Monitor ();

6a)   Neuen Zeitpunkt t^ bestimmen und Zeit fortschalten
      --------------------------------------------------
      t^ =  ... ;
      t  =  t^;

6b)   Alle Zustandsvariablen umspeichern
      ----------------------------------
      Fuer alle i aus {1..NZ}:
      z_i =  z_i^;
   }
   Ende Schleife ueber alle Zeitpunkte
   ===================================
```

Es werden nur die Werte solcher Sensorvariablen berechnet, die nicht mit internen Variablen anderer Komponenten verbunden sind. Da die nicht angeschlossenen Sensorvariablen nur von der Zeit abhängig sind, ist ihre Berechnung nur in der ersten Ebene erforderlich.

Die abhängigen Variablen werden `NE-1` mal immer wieder aufs neue bestimmt. Das Sortieren der Gleichungen ist bei dieser Vorgehensweise daher nicht erforderlich.

Die Zustandsvariablen für den Folgezeitpunkt werden abschließend berechnet, wenn die Werte der übrigen Variablen mit Sicherheit schon bestimmt sind.

Der gezeigte Algorithmus gliedert sich in gleichbleibende Abschnitte und einen Abschnitt, der für einzelne Komponenten steht. Die gleichbleibenden Abschnitte (Ablaufsteuerung) sind im Laufzeitsystem untergebracht, die komponentenspezifischen Abschnitte werden vom Compiler aus der Modellbeschreibung (Spezifikation) erzeugt. Für Komponenten der gleichen Klasse liegt nur eine Codierung vor. Abbildung 2.7-1 zeigt die Struktur eines in Komponenten zerlegten Simulationsprogramms.

Der Aufwand zur Berechnung der abhängigen Variablen ist jedoch beträchtlich: Insgesamt werden hierzu pro Zeitpunkt

`(NE-1) * NY` statische Beziehungen berechnet und
`(NE-1) * NK` Unterprogrammaufrufe ausgeführt.

Dieser hohe Mehraufwand gegenüber der Bearbeitung in einem einzigen Programmstück ist natürlich nicht mehr akzeptabel. Wir wollen daher verschiedene Möglichkeiten ausloten, hier Verbesserungen herbeizuführen.

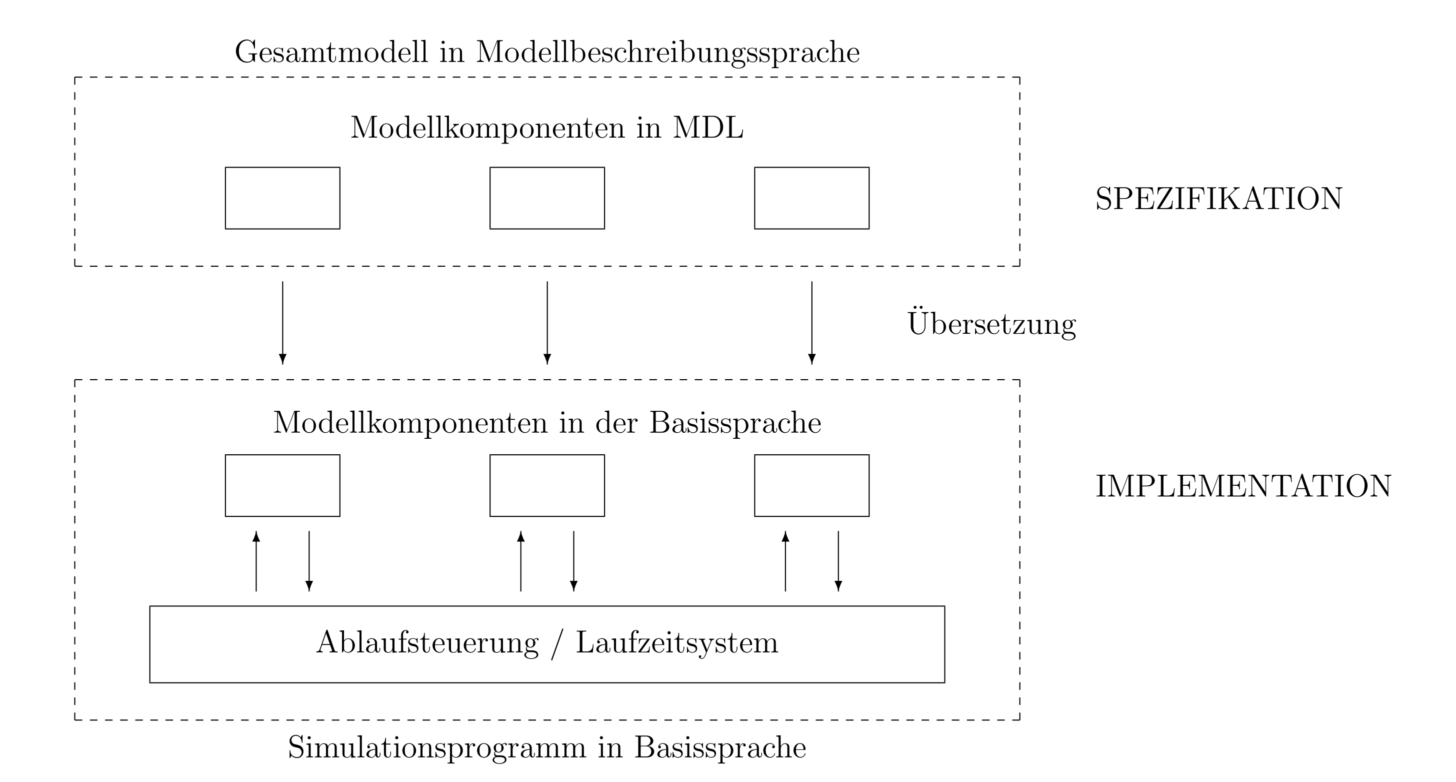



Abb. 2.7-1: Erzeugung eines modularen Simulationsprogramms

Die einfachste Lösung beruht auf einem vorzeitigen Abbruch der Iteration über alle Ebenen: In der Regel kann man davon ausgehen, daß die statischen Beziehungen teilweise bereits sortiert sind und sich daher die korrekten Werte der abhängigen Variablen bereits bei weniger Durchläufen einstellen. Sobald man feststellt, daß ein Durchlauf keine Änderungen von abhängigen Variablen mehr bewirkt hat, kann man die Iteration beenden und mit der Bestimmung des neuen Zustands fortfahren.

Andererseits zeigt die Erfahrung, daß man gerade bei größeren Modellen durchaus mit mindestens 5 Zyklen rechnen muß, da die Modellkomponenten oft in einer unglücklichen Reihenfolge aufgerufen werden. Wir sind daher mehr an gezielten Maßnahmen interessiert, die auch von der Einflußnahme des Benutzers unabhängig sind.

Für Effizienzverbesserungen werden wir wiederum Marken einsetzen. Es ist die Aufgabe der Marken zu steuern, welche Teile des Programms tatsächlich auszuführen sind. Hierbei unterscheiden wir Maßnahmen, die mit statischen Marken arbeiten von solchen, die dynamische Marken einsetzen. Statische Marken legen den Programmfluß einmal fest und werden über den ganzen Programmlauf hinweg beibehalten. Dynamische Marken optimieren den Programmfluß während des Ablaufs.

### 2.7.2 Effizienzverbesserung durch statische Marken

Alle Effizienzverbesserungen beruhen im wesentlichen darauf, eine Komponente nur dann aufzurufen und eine Variable nur dann neu zu berechnen, wenn es nötig ist.

Mit Hilfe statischer Marken läßt sich dies durch die beiden folgenden Maßnahmen erreichen:

1. Aussondern derjenigen Bestimmungsgleichungen, die in der aktuellen Ebene nicht zu berechnen sind.

Hierzu erhält jede Variable neben ihrem Wert als weiteren Eintrag die Nummer der Ebene (y_i.Level) zugeordnet, in der ihr Wert zu berechnen ist. Bevor die Beziehung ausgewertet wird, wird auf die richtige Ebene geprüft. Zur Berechnung der abhängigen Variablen sind nunmehr für jeden Zeitpunkt

| | |
|---|---|
| NY | statische Beziehungen zu berechnen, |
| (NE-1) * NK | Unterprogramme aufzurufen, |
| (NE-1) * NK * NY | Abfragen der Marken auszuführen. |

2. Aussondern derjenigen Komponenten, die keine Variablen enthalten, welche in der aktuellen Ebene zu berechnen sind.

   Es wird ein zweidimensionales Feld B mit logischen Werten der Dimension [NK] [NE] angelegt. Ein Feldelement B [j][m] enthält eine Marke, wenn die Komponente j Bestimmungsgleichungen enthält, welche in der Ebene m auszuführen sind.

   Die Anzahl der aufzurufenden Unterprogramme verringert sich hierdurch erfahrungsgemäß etwa um den Faktor 2 bis 3.

Der Grund, warum diese Bedingungen nicht in den Code aufgenommen werden können, sondern als Daten in Form statischer Marken abzulegen sind, liegt im Klassenkonzept. Eine Variable, die in der einen Ausprägung einer Klasse in Ebene i berechnen ist, ist in einer anderen Ausprägung möglicherweise in Ebene j zu bestimmen.

Der Grundalgorithmus ist hierzu durch die eingeführten Marken zu ergänzen:

```
1)  Anfangszeit und Anfangszustand setzen
    -------------------------------------
    t = t_0
    Z = Z_0;

    Schleife ueber alle Zeitpunkte
    ==============================
    {
       Schleife ueber alle Ebenen
       ==========================
       Fuer alle m von 1 .. NE:
       {
          Schleife ueber alle Komponenten
          ===============================
          Fuer alle j aus {1..NK}:
          {
             Falls B [j][m] = TRUE:

             Aufruf der Komponente j
             =======================
             {
2)              Alle nicht verbundenen Sensorvariablen x_i berechnen
                ---------------------------------------------------
                Fuer alle nicht verbundenen x_(i,j) mit i aus {1..NX_j}:
                  Falls Ebene m = 1:
                  x_(i,j) =  e_(i,j) (t);
```

```
3)              Alle abhaengigen Variablen y_i berechnen
                ----------------------------------------
                Fuer alle i aus {1..NY_j}:
                  Falls Ebene m = y_(i,j).Level:
                  y_(i,j) =  g_(i,j) (X_j, Y_j, Z_j, t);

4)              Alle neuen Zustandsvariablen z_i^ berechnen
                -------------------------------------------
                Fuer alle i aus {1..NZ_j}:
                  Falls Ebene m = NE:
                  z_(i,j)^ =  f_(i,j) (X_j, Y_j, Z_j, t);
             }
             Ende des Komponentenaufrufs
             ===========================
          }
          Ende Schleife ueber alle Komponenten
          ====================================
       }
       Ende Schleife ueber alle Ebenen
       ===============================

5)     Variablen des aktuellen Zustands aufzeichnen
       --------------------------------------------
       Monitor ();

6a)    Neuen Zeitpunkt t^ bestimmen und Zeit fortschalten
       --------------------------------------------------
       t^ =  ... ;
       t  =  t^;

6b)    Alle Zustandsvariablen umspeichern
       ----------------------------------
       Fuer alle i aus {1..NZ}:
       z_i =  z_i^;
    }
    Ende Schleife ueber alle Zeitpunkte
    ===================================
```

Die abhängigen Variablen werden nun zu einem Zeitpunkt nur noch ein einziges Mal bestimmt.

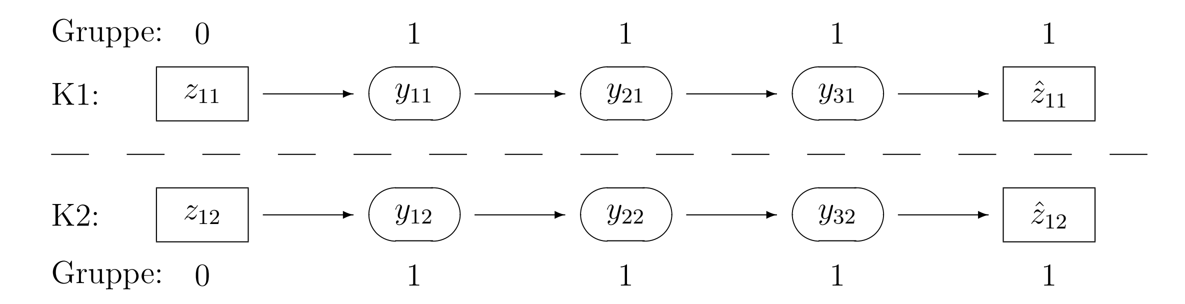
Dieser Algorithmus bleibt für alle weiteren Verbesserungen der gleiche. Es ist nur die Frage, wie man durch geschicktes Setzen der Marken und durch eine Verminderung der Anzahl der Ebenen die Zahl der auszuführenden Anweisungen weiter reduzieren kann.

Die Reduktion der Anzahl der Ebenen läßt sich erreichen, wenn man die Gleichungen innerhalb einer Komponente sortiert und in Gruppen zusamenfäßt. Auf diese Maßnahme, die auch mit statischen Marken durchgeführt wird, soll nun näher eingegangen werden.

**Reduktion der Anzahl der Ebenen**

Sortierte Gleichungen in einer Komponente können zu einer Gruppe zusammengefaßt und hintereinander weg ausgeführt werden, wenn zu ihrer Bestimmung nicht über Sensorvariablen auf Variablen fremder Komponenten zugegriffen werden muß. Variablen einer Gruppe werden in der gleichen Ebene ausgeführt. Dadurch vermindert sich in der Regel die Zahl der Komponentenaufrufe.

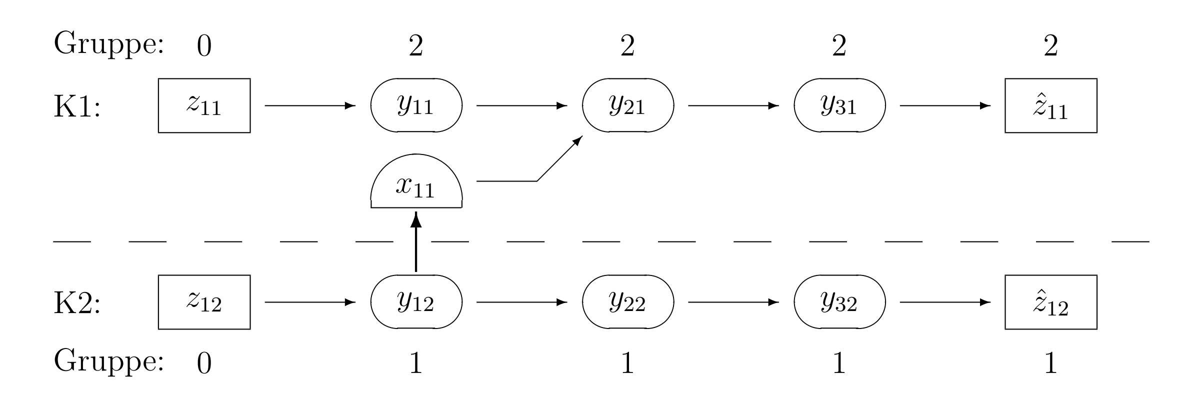
Beispiel:




Sind die Gleichungen sortiert, können sie zu einer einzigen Gruppe zusammengefaßt werden, d.h. alle Berechnungen können in einer Ebene ausgeführt werden. Man kommt mit zwei Komponentenaufrufen aus.

Werden Werte von Variablen anderer Komponenten benötigt, dann müssen erst diese Werte ermittelt werden, bevor mit der Berechnung begonnen bzw. fortgefahren werden kann. Um die korrekte Reihenfolge einzuhalten, sind die importierten Variablen einer anderen Gruppe zuzuordnen und vorher zu berechnen.

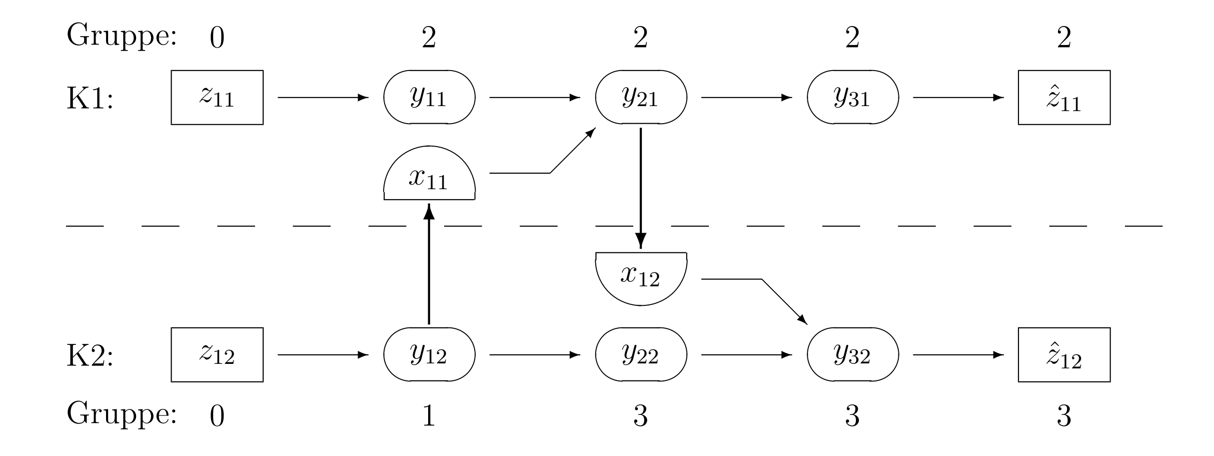
Beispiel: Einwirkung einer anderen Komponente




Die optimale Bearbeitung besteht darin, daß die abhängigen Variablen der Komponente K2 in der Gruppe 1, die der Komponente K1 in der Gruppe 2 ausgeführt werden. Auch hier kommt man mit zwei Komponentenaufrufen aus.

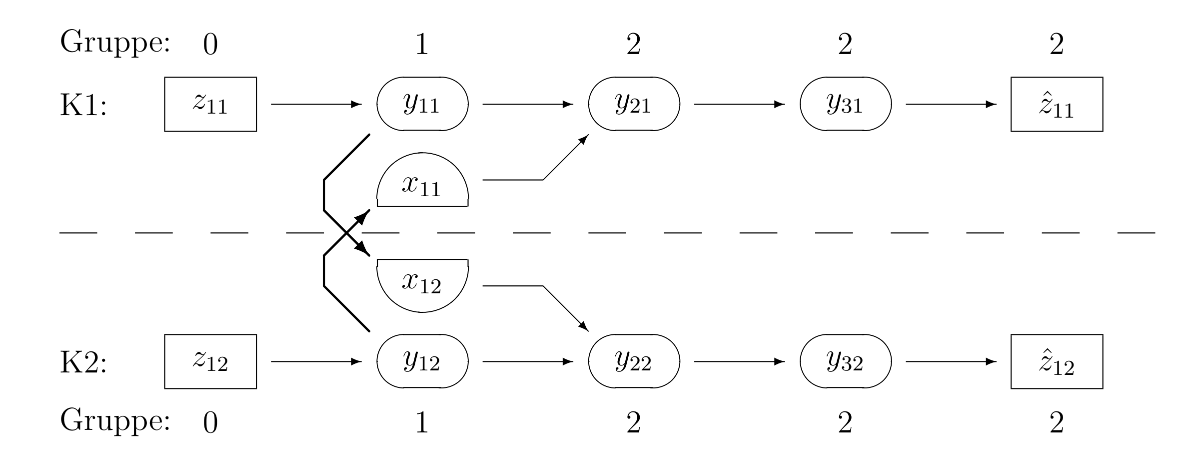
Bestehen Rückwirkungen oder Wechselwirkungen, dann können die internen Variablen einer Komponente nicht mehr gemeinsam in einer Gruppe berechnet werden. In diesem Fall muß das Unterprogramm der Komponente mehrfach betreten werden.

Beispiel: Rückwirkung




Hier besteht die optimale Bearbeitung darin, zunächst die abhängige Variable $y_{12}$ in Gruppe 1 zu berechnen, daraufhin alle abhängigen Variablen der Komponente 1 in Gruppe 2 und schließlich $y_{22}$ und $y_{32}$ in Gruppe 3. Die Variable $y_{22}$ hätte auch in Gruppe 1 berechnet werden können. Es sind drei Komponentenaufrufe nötig.

Beispiel: Wechselwirkung




Hier besteht die optimale Bearbeitung darin, zunächst die abhängigen Variablen $y_{11}$ und $y_{12}$ in Gruppe 1 zu berechnen und daraufhin alle übrigen abhängigen Variablen in Gruppe 2. In diesem Fall braucht man vier Komponentenaufrufe.

Wie man sieht, kann man durch eine geeignete Gruppierung die Anzahl der benötigten Komponentenaufrufe erheblich reduzieren.

Die Abwicklung zur Laufzeit erfolgt mit Hilfe des bereits angegebenen Algorithmus. Nur tritt an die Stelle der Schleife über alle Ebenen eine Schleife über alle Gruppen.

Bei der Vergabe der Gruppennummer sind zwei Gesichtspunkte zu beachten:

1. Angeschlossene Sensorvariablen erhalten immer eine um eins höhere Gruppennummer als die Variable, mit der sie verbunden ist.
   Gruppe $(x)$ = Gruppe $(v) + 1$; wenn $x \equiv v$

2. Zustandsvariablen, abhängige Variablen und nicht angeschlossene Sensorvariablen übernehmen die größte Gruppennummer von denjenigen Variablen, von denen sie abhängen. In manchen Fällen ist auch eine (noch) größere Gruppennummer sinnvoll.
   Gruppe $(z) \geq$ Max (Gruppe$(v_1)$, ... , Gruppe$(v_n)$)

Der Weg im Abhängigkeitsgraph, der zu einer Variable über die meisten Sensorvariablen führt, liefert die insgesamt größte Gruppennummer.

**Algorithmus für die Gruppierung von algebraischen Gleichungen**

Diese Vorgaben lassen aber noch Freiheiten für die Zuordnung der Gruppennummer an die Variablen und es stellt sich daher die Frage, mit welchem Algorithmus eine optimale Lösung erzielt wird. Optimal bedeutet in diesem Zusammenhang, daß die Anzahl der Komponentenaufrufe für jeden Zeitpunkt minimal gehalten wird. Dadurch spart man nicht nur den Unterprogammaufruf ein, sondern vor allem das aufwendige Überprüfen der Bedingungen für jede Variable.

Wir wollen uns an dieser Stelle mit der Beschreibung einer nicht ganz optimalen Vorgehensweise begnügen, die aber bereits ausreichende Ergebnisse liefert.

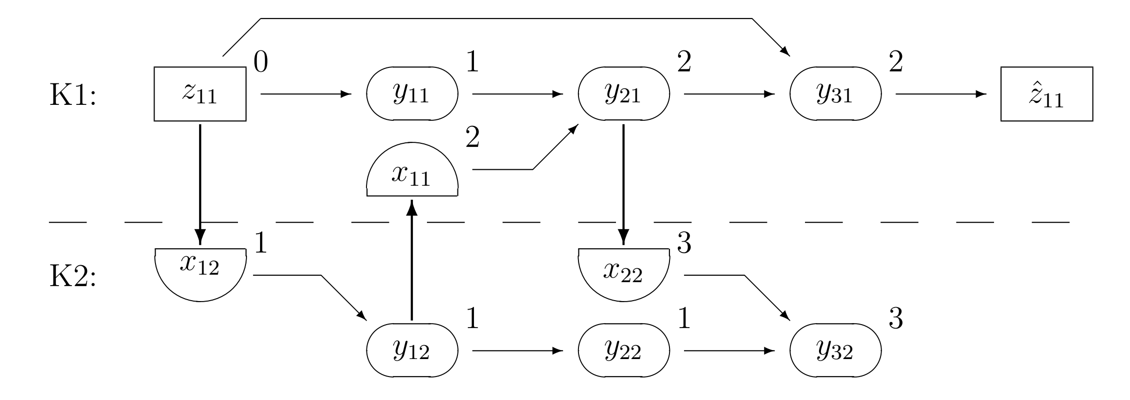
Knoten, die nur von Zustandsvariablen oder der Zeit abhängen, erhalten die Gruppennummer 1 eingetragen. Wir verfolgen nun jeden Pfad des Graphen und tragen die Nummer des vorangegangenen Knotens ein. Hängt ein Knoten von mehreren anderen Knoten ab, dann wird die jeweils größte Gruppennummer übernommen. Für Sensorvariablen tragen wir die um 1 erhöhte Gruppennummer der angeschlossenen Variablen ein.

Bei jedem Wechsel des Signalpfads auf eine andere Komponente wird demnach die Gruppennummer um 1 erhöht. Die Gruppennummer jeder Variable zeigt nun an, über wie viele Sensorvariablen der Pfad im Abhängigkeitsgraphen maximal führt, d.h. wie häufig zwischen Komponenten zu wechseln ist.

Beispiel:



$$
\begin{aligned}
y_{11} &:= g_{11}(z_{11}); & y_{12} &:= g_{12}(x_{12});\\
y_{21} &:= g_{21}(x_{11}, y_{11}); & y_{22} &:= g_{22}(y_{12});\\
y_{31} &:= g_{31}(z_{11}, y_{21}); & y_{32} &:= g_{32}(x_{22}, y_{22});
\end{aligned}
$$




Eine weitere Optimierung wäre möglich, würde man $y_{11}$ der Gruppe 2 zuordnen.

**Abschätzung des Rechenaufwands**

Durch Kombination der drei Maßnahmen sind nunmehr für jeden Zeitpunkt zur Bestimmung der abhängigen Variablen

`NY` statische Beziehungen zu berechnen,
`NG * b * NK` Unterprogramme aufzurufen,
`NG * b * NY` Abfragen auszuführen,

wenn man davon ausgeht, daß die Anzahl der Gruppen `NG` beträgt und pro Gruppe `b * NK` Komponenten aufzurufen sind.

Geht man - typischen Erfahrungen gemäß - von einer Gruppenanzahl von 3 aus und davon, daß mit jeder Gruppe durchschnittlich zwei von drei Komponenten `(b=2/3)` aufzurufen sind, ergeben sich

`2 * NK` Unterprogrammaufrufe und
`2 * NY` Abfragen.

Macht man vom Setzen statischer Marken Gebrauch, muß man für die Siumulation eines modular aufgebauten Modells gegenüber einer Realisierung in einer einzigen Komponente trotzdem noch mit einem bis zu doppelt so hohem Aufwand rechnen.

### 2.7.3 Effizienzverbesserung durch dynamische Marken

Beim Einsatz dynamischer Marken wird während des Programmablaufs geprüft, ob eine Komponente aufzurufen bzw. innerhalb einer Komponente eine Variable zu berechnen ist. Es lassen in diesem Fall drei Verfahren unterscheiden:

1. Markierung der Auswirkungen einer Änderung (Vorwärts-Verfolgung):
   - Komponentenaufruf, falls Komponente und Ebene/Gruppe markiert
   - Berechnung, falls Variable markiert

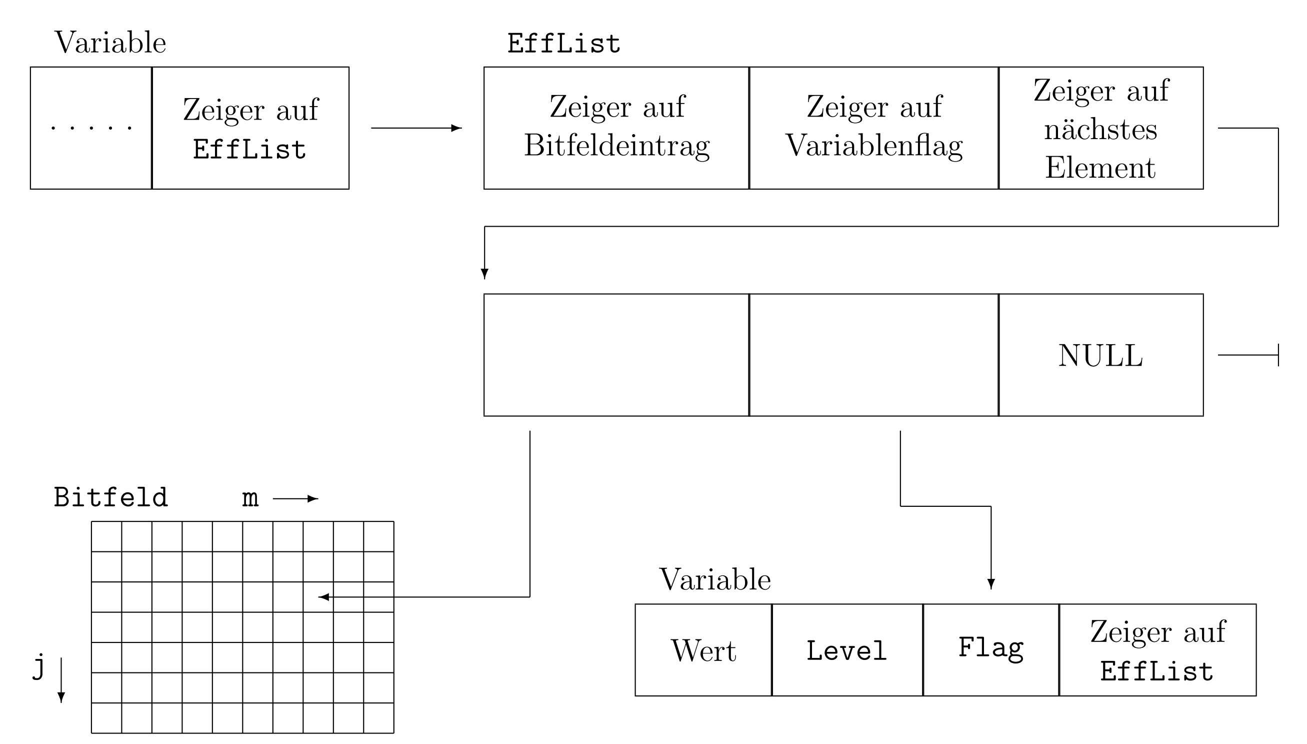
2. Markierung der veränderten Variablen (Rückwärts-Verfolgung):
   – Komponentenaufruf, falls Abhängigkeit einer internen Variablen markiert
   – Berechnung, falls Abhängigkeit einer Variablen markiert

3. Markierung der Auswirkungen auf Komponente/Gruppe und Markierung der veränderten Variablen (gemischte Lösung):
   – Komponentenaufruf, falls Komponente und Ebene/Gruppe markiert
   – Berechnung, falls Abhängigkeit einer Variablen markiert

### Vorwärts-Verfolgung

Eine Marke im Bitfeld `B` für Ebene und Komponente bzw. die Marke einer Variablen werden dann gesetzt, wenn wenn nach einer Wertänderung ein Einfluß auf die betreffende Komponente bzw. Variable vorliegt. Für diesen Zweck ist zu jeder Variablen zu vermerken, auf welche anderen Komponenten bzw. Variablen sie Einfluß ausübt.

Dazu legt man für jede Variable eine Auswirkungsliste an. Sie enthält für jeden Pfad im Abhängigkeitsgraphen, der von der betreffenden Variable ausgeht, ein Listenelement. In jedem Listenelement befindet sich je ein Zeiger auf einen Eintrag im Bitfeld `B` und auf die Marke `Flag` der Variable. Der Eintrag `Level` gibt die Gruppenzugehörigkeit einer Variablen an.




Nach der Wertänderung einer Variablen werden für jedes Element der Auswirkungsliste die Marken gesetzt:

| | |
|---|---|
| Erstes Listenelement: | `EffEntry = <Variable>.EffList;` |
| Besetzung des Bitfeldes: | `EffEntry->Zeiger_BitFeld = TRUE;` |
| Besetzung des Variablenflags: | `EffEntry->Zeiger_VarFlag = TRUE;` |
| Nächstes Listenelement: | `EffEntry = EffEntry->Zeiger_EffList;` |

Es fällt auf, daß das Setzen der Marken nahezu keine Rechenzeit in Anspruch nimmt. Dagegen erfordert die Auswirkungsliste und die Vorbesetzung der Auswirkungsliste einigen Speicherplatz. Dies stellt jedoch bei den heutzutage sehr geringen Kosten im allgemeinen kein Problem dar.

Der Algorithmus für die Simulation ist nun noch durch die Abfrage der Variablenflags zu ergänzen. Vor der Berechnung einer Bestimmungsgleichung wird zusätzlich zur Ebene/Gruppe auch geprüft, ob überhaupt ein Einfluß auf die zu setzende Variable vorliegt. Für die abhängigen Variablen sieht dies beispielsweise so aus:

```
3)  Alle abhaengigen Variablen y_i berechnen
    ----------------------------------------
    Fuer alle i aus {1..NY_j}:
      Falls Ebene  m = y_(i,j).Level:
        Falls  y_(i,j).Flag = TRUE:
        y_(i,j) =  g_(i,j) (X_j, y_(1,j), ... , y_(i-1,j), Z_j, t);
        Falls Wertaenderung:  Setzen der Marken (siehe oben)
```

Bei der dynamischen Markierung sind also für jeden Zeitpunkt

| | |
|---|---|
| `c * NY` | Ausgangsbeziehungen zu berechnen, |
| `c * NG * b * NK` | Unterprogramme aufzurufen, |
| `c * NG * b * NY` | Abfragen auszuführen, |

wenn man davon ausgeht, daß die Anzahl der Gruppen `NG` beträgt, pro Ebene `b * NK` Komponenten aufzurufen sind und pro Zeitpunkt nur `c * NY` statische Beziehungen berechnet werden müssen. Da sich die Auswirkungen auf wenige Komponenten beschränken, geht die Anzahl der Unterprogrammaufrufe und der Abfragen entsprechend zurück.

Bei einer Realisierung in einer Komponente müßten pro Zeitpunkt `NY` Abfragen ausgeführt werden. Für größere Modellen, bei denen `c` deutlich kleiner als 1 wird, ergibt sich bei modularer Zerlegung sogar ein geringerer Aufwand. Dies rührt daher, weil pro Zeitpunkt nur wenige kleine Komponenten anstelle einer sehr großen zu durchlaufen sind.

### Rückwärts-Verfolgung

Bei diesem Verfahren braucht man den Abhängigkeitsgraph nicht durch entsprechende Datenbereiche zu repräsentieren. Für jede Variable wird lediglich durch Setzen der Marke `Flag` vermerkt, ob sie verändert wurde. Anhand dieser Flags wird geprüft, ob eine Komponente zu durchlaufen ist bzw. ob eine Bestimmungsgleichung auszuwerten ist.

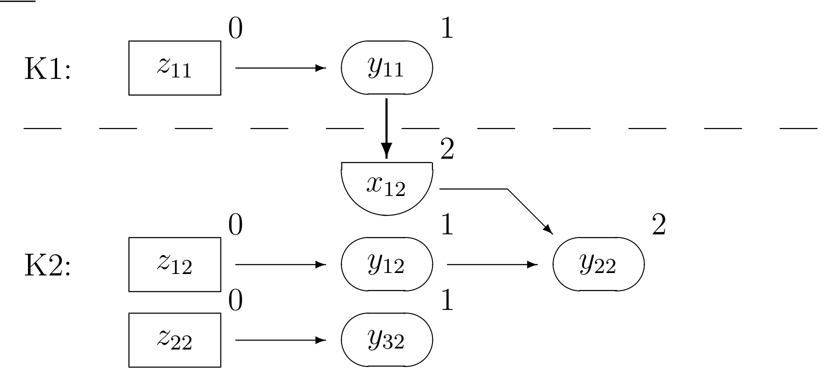
Eine Komponente wird nur durchlaufen, wenn sich interne Variablen oder Sensorvariablen der Komponente verändert haben, die auf Variablen der aktuellen Gruppe Einfluß nehmen.

Eine Bestimmungsgleichung in einer Komponente wird nur dann ausgeführt, wenn

- sie der aktuell auszuführenden Gruppe angehört <u>und</u>
- Variablen, von denen diese Gleichung abhängt ihren Wert verändert haben.

Die Prüfung, ob eine Komponente zu durchlaufen ist, ist sehr aufwendig.

Beispiel:




Die Komponente K2 ist zu durchlaufen, wenn

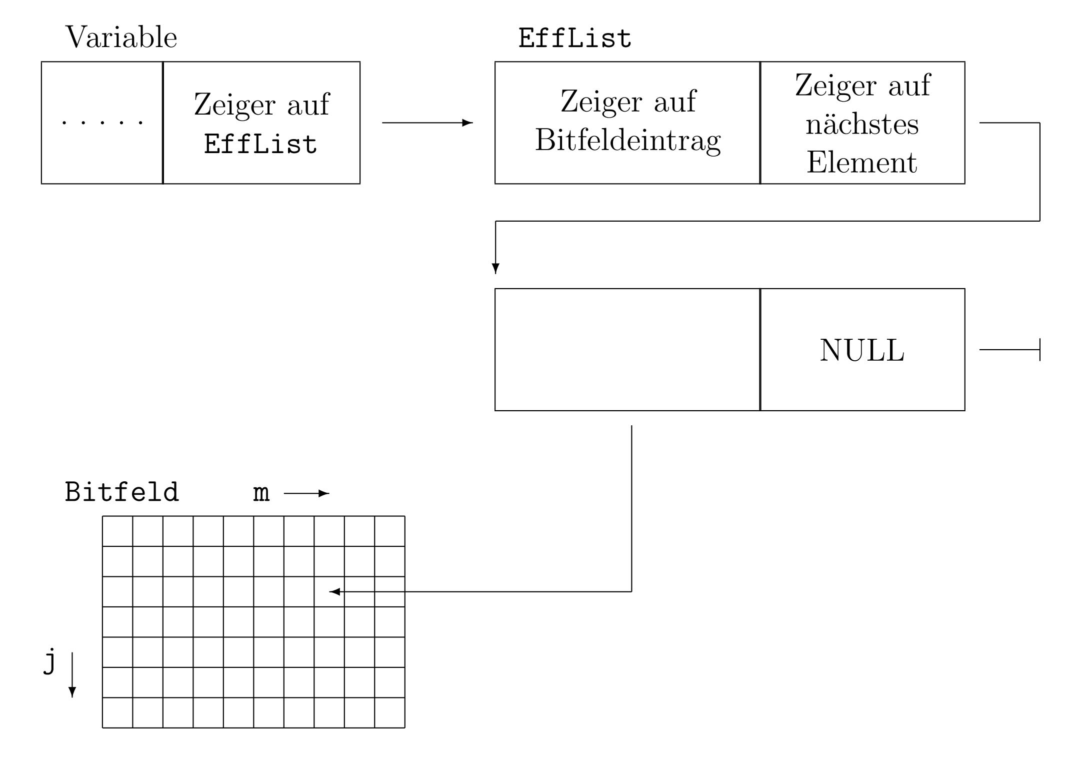
- die Gruppennummer gleich 1 ist und sich $z_{12}$ oder $z_{22}$ verändert hat
- die Gruppennummer gleich 2 ist und sich $y_{12}$ oder $x_{12}$ verändert hat.

Dieselben Abfragen, die innerhalb der Komponente nötig sind, werden vor dem Betreten der Komponente nochmals erforderlich. Daraus ist ersichtlich, daß es mit diesem Verfahren nicht möglich ist, auf effiziente Art und Weise das Durchlaufen einer Komponente zu unterbinden.

### Kombinierte Vorwärts- und Rückwärts-Verfolgung

Von Interesse ist aber eine Kombination der beiden Verfahren. Durch Vorwärts-Verfolgung wird markiert, auf welche Komponenten die Veränderung einer Variablen Einfluß hat. Durch Rückwärts-Verfolgung wird innerhalb einer Komponente festgestellt, ob eine Bestimmungsgleichung auszuführen ist.

Gegenüber der reinen Vorwärts-Verfolgung verkürzen sich die Abhängigkeitslisten ganz entscheidend.



Bisher mußte für jede Auswirkung auf eine andere Variable, d.h. für jeden Pfad im Abhängigkeitsgraphen ein Listenelement angelegt werden. Nun muß nur noch für eine Auswirkung auf eine andere Komponente/Gruppe ein Listenelement angelegt werden. Viele Variablen besitzen dadurch überhaupt keine Abhängigkeitsliste mehr.

Für Auswirkungen auf die aktuelle Komponente/Gruppe ist kein Eintrag mehr erforderlich. Mehrere Auswirkungen auf die gleiche Komponente/Gruppe können in einem gemeinsamen Listenelement zusammengefaßt werden.

Die Berechnung der abhängigen Variablen gestaltet sich demnach wie folgt:

```
3)  Alle abhaengigen Variablen y_i berechnen
    ----------------------------------------
    Fuer alle i aus {1..NY_j}:
      Falls Ebene  m = y_(i,j).Level:
        Falls       x_(p1,j).Flag  und  x_(p2,j).Flag  und  ...  und
                y_(q1,j).Flag  und  y_(q2,j).Flag  und  ...  und
                z_(r1,j).Flag  und  z_(r2,j).Flag  und  ...  und  t.Flag :
        {
          y_(i,j) =  g_(i,j) (x_(p1,j), x_(p2,j), ... ,
                              y_(q1,j), y_(q2,j), ... ,
                              z_(r1,j), y_(r2,j), ... , t);

          Falls Wertaenderung:  * Abhaengigkeitsliste verfolgen und
                                  Bitfeldeintraege vornehmen
                                * y_(i,j).Flag = TRUE;
        }
```

Sensorvariablen übernehmen zu diesem Zweck die Änderungsmarke derjenigen Variable, auf die sie verweisen. Die Vorbesetzung und das Rücksetzen der Marken erfolgt wie bereits besprochen.

Die Ausführungen haben gezeigt, daß sich auch bei einer Modularisierung des Modells der Aufwand zur Berechnung sehr stark herabsetzen läßt und ungefähr einer Realisierung in einer einzigen Komponente gleichkommt. Eine effiziente Bearbeitung einer expliziten Modelldarstellung ist demnach auch bei modularer Modellzerlegung sichergestellt.

## 2.8 Zusammenfassung

In diesem Kapitel wurde gezeigt, daß die systemtheoretische Modelldarstellung als Grundlage für eine Spezifikationssprache zur Beschreibung dynamischer Modelle dienen kann. Da die Zielrichtung im Bereich diskreter Modelle liegt, beschränkten wir uns auf die explizite Darstellungsform.

Als besondere Vorteile ergaben sich:

- Die Modellbeschreibungssprache ist deklarativ und entbindet den Anwender von der Kontrolle des Programmflusses, d.h. die Reihenfolge, in der die Aussagen stehen, hat keinen Einfluß auf die Modellsemantik.
- Die Sprache läßt weitgehende Überprüfungen der Aussagen auf semantische Korrektheit zu. Auf diese Weise kann dem Anwender ein hohes Maß an Sicherheit gewährleistet werden.
- Die Sprache erlaubt eine effiziente Umsetzung in eine prozedurale Sprache zur algorithmischen Abarbeitung einer Modellbeschreibung.
- Die Sprache stellt ein einfaches Konzept für eine hierarchische Modularisierung eines Modells zur Verfügung. Dabei müssen weder Einbußen bei der Überprüfung der Modellsemantik noch bei der effizienten Bearbeitung hingenommen werden. Die Modellstruktur läßt sich sehr gut graphisch repräsentieren.

Der Sprachaufbau umfaßt:

- Basiskomponenten zur Beschreibung der Modelldynamik und höhere Komponenten zur Beschreibung der Modellstruktur.
- Basiskomponenten beinhalten einen Deklarationsteil zum Vereinbaren der Modellvariablen mit ihren Eigenschaften und einen Dynamikteil zum Definieren der Modellvariablen. Die Definition legt den Wert einer Variablen in Abhängigkeit der anderen Modellvariablen zu jedem Zeitpunkt fest.
- Höhere Komponenten beinhalten die Deklaration der Subkomponenten, Deklaration einer Schnittstelle mit Sensorvariablen und internen Variablen aus Subkomponenten, Verbindungen zwischen Subkomponenten und die Initialisierung der Variablen aus Subkomponenten.
- Modellvariablen wurden unterschieden in Zustandsvariablen, abhängige Variablen und Sensorvariablen.
- Für die Zustandsüberführungsfunktion und die Zeitmenge wurden verschiedene Möglichkeiten aufgezeigt. Wir werden hier stets die reellen Zahlen als Modellzeit heranziehen. Auf die Besonderheiten bei der Implementierung, die sogenannte Implementierungszeit, wurde hingewiesen.

Im folgenden wird die Sprache weiter ausgebaut, um auch Warteschlangen- und Transportmodelle beschreiben zu können. Dabei wird versucht, die genannten Vorteile dieser Sprache zu erhalten.

# Kapitel 3

# Spracherweiterungen zur Formulierung von Warteschlangenmodellen

In diesem Kapitel wird erörtert, wie Warteschlangenmodelle zustandsorientiert beschrieben werden können. Hierzu ist der bisherige Ansatz um eine ganze Reihe sprachlicher Mittel zu erweitern. Dabei wird zunächst von naheliegenden Ergänzungen ausgegangen, wie sie das erste einführende Beispiel zeigt.

Diese Spracherweiterungen werden diskutiert und es wird versucht, deren Probleme durch weitere Ergänzungen des Sprachkonzepts zu überwinden. Auf diese Weise ergibt sich nach und nach ein beträchtlicher Sprachumfang. Glücklicherweise wird schließlich eine Lösung gefunden, die einen großen Teil der bereits eingeführten Sprachkonstrukte wieder überflüssig macht.

Dieser Umweg wird bewußt beschritten, weil die gefundene Lösung nicht auf der Hand liegt und das systemtheoretische Konzept um neue Inhalte erweitert. Dies wird besser verständlich, wenn der Bezug zum ursprünglichen Konzept hergestellt wird. Durch die allmähliche Erweiterung der Sprache erkennt der Leser auch gleichzeitig die Intention, mit der die neuen Konstrukte und Datentypen eingeführt werden.

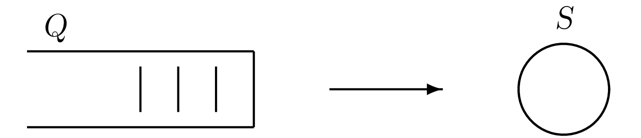
Das im vorangegangenen Kapitel dargestellte Grundkonzept und die Überlegungen für seine algorithmische Umsetzung gelten auch für die vorzunehmenden Spracherweiterungen.

Besonderer Wert wird auch weiterhin darauf gelegt, daß die semantische Korrektheit vom Compiler überprüft werden kann. Dieser Gesichtpunkt ist deshalb so wichtig, weil er die beim Sprachentwurf bestehenden Freiheiten sinnvoll einschränkt.

Als wichtigste Anforderung überhaupt wird allerdings die Modularisierbarkeit von Warteschlangen- und Transportmodellen angesehen.

## 3.1 Motivation

Wir betrachten den folgenden typischen Ausschnitt aus einem Warteschlangensystem.




Aufträge vom Typ `Job` in einer Warteschlange `Q` warten auf ihre Bearbeitung in der Bedieneinheit `S`. Sobald die Bedieneinheit frei ist, darf der erste Auftrag aus der Warteschlange in die Bedieneinheit wechseln. Die in der Warteschlange verbliebenen Aufträge rücken dann um eine Position nach vorne.

In einem ersten Ansatz könnte eine formale Beschreibung etwa folgendermaßen aussehen:

```
MOBILE COMPONENT  Job

DEFINITIONS
   VALUE SET  Queues: ('Q1', 'Q2', 'Q3')

DECLARATIONS
   STATE VARIABLES
   DISCRETE   Ort (Queues)  :=  'Q1' ,
              Pos (INT)     :=   1
END OF  Job

ARRAY {1..N}  Job

FOREACH  i  IN  1..N
LET
    IF  Job[i].Ort = 'Q'  AND  EMPTY {j IN 1..N | Job[j].Ort = 'S'}
    LET
        IF  Job[i].Pos = 1
        LET
            Job[i].Ort^ :=  'S';
        ELSE
        LET
            Job[i].Pos^ :=  Job[i].Pos - 1;
        END
    END
END
```

Bereits dieses einfache Modell macht eine Reihe von Erweiterungen notwendig:

Zu einem Auftrag gehören die Attribute `Ort` und `Pos`. Um diese Verbindung deutlich zu machen, sind Verbunde von Attributen erforderlich.

Damit die Modellbeschreibung nicht für jeden einzelnen Auftrag erstellt werden muß, werden Felder von Modellelementen und eine Art Allquantor (`FOREACH`) eingeführt.

Es muß weiterhin festgestellt werden, ob sich ein Auftrag in der Bedieneinheit befindet, es also einen Auftrag gibt, dessen Attribut `Ort` den Wert `S` besitzt. Hierfür ist eine Mengenbildung oder zumindest eine Funktion zum Feststellen der Kardinalität einer Menge erforderlich.

Das Beispiel zeigt auch bereits Probleme, die typisch für die weiteren Ausführungen dieses Kapitels sind.

Die Vorbesetzungen der Zustandsvariablen gelten für alle Ausprägungen der Klasse `Job`. Hieraus resultiert, daß zunächst alle mobilen Elemente den gleichen Ort und die gleiche Position besitzen. Zumindest aber die Position muß individuell vorbesetzt werden.

Von einer Zustandsvariable wie dem Attribut `Pos` erwartet man normalerweise, daß sie zu jedem Zeitpunkt jeden beliebigen Wert aus ihrer Wertemenge annehmen kann. Dies ist offensichtlich nicht der Fall und wir haben an dieser Stelle bereits einen ersten Hinweis, daß die Beschreibung von Warteschlangen über ein Positions-Attribut nicht der geeignete Weg ist.

Bevor wir einzelne Sprachkonstrukte einführen, wollen wir klären, welche Anforderungen damit abgedeckt werden sollen.

## 3.2 Anforderungen

Bereits in der Einleitung wurden die Anforderungen an die Funktionalität einer Modellbeschreibungssprache erörtert. Diese Anforderungen wurden einer ganz konkreten Anwendung entlehnt. Da wir eine universell einsetzbare Sprache anstreben, soll an dieser Stelle festgestellt werden, welche Anforderungen sich aus theoretischen Überlegungen ergeben, wenn man versucht, mit folgender gedanklicher Modellvorstellung zu operieren:

Warteschlangen- und Transportmodelle bestehen aus einer größeren Anzahl von beweglichen Einheiten, die sich durch einen Verbund von Attributen auszeichnen. Aufgrund bestimmter Eigenschaften können solche beweglichen Einheiten zu Mengen zusammengefaßt werden.

Eine Mengenbildung erfolgt in der Regel, um Einheiten, die sich am gleichen (physikalischen oder logischen) Aufenthaltsort befinden, zusammenzufassen. Eine solche Menge kann man sich geordnet (Warteschlange) oder ungeordnet (Warteraum) vorstellen.

Für die Dynamik eines Modells ist es nun wichtig festzustellen, wieviele bewegliche Einheiten mit bestimmten Eigenschaften sich an einem Aufenthaltsort befinden. Ebenso ist es wichtig, daß bewegliche Einheiten mit bestimmten Eigenschaften aus einem Warteraum selektiert und verändert werden können.

Um mit der Vorstellung eines Warteraums arbeiten zu können, muß es möglich sein,

- Eigenschaftsmengen von beweglichen Einheiten zu bilden
- die Anzahl von Elementen dieser Menge (Kardinalität) festzustellen
- auf einzelne Elemente dieser Mengen selektiv zuzugreifen

In der Regel enthält eine bewegliche Einheit ein Attribut, das für den Aufenthaltsort steht und das zur Mengenbildung herangezogen werden kann.

Die Selektion eines Elements aus einer ungeordneten Menge kann nur durch die Angabe von Selektionkriterien erfolgen, die quasi auf ein Ordnen der Menge hinauslaufen (z.B höchste Priorität).

Die Attribute selektierter Einheiten können dann entweder gelesen oder beschrieben werden. Durch ein Verändern des Aufenthaltsorts ist auch ein Transport einer (selektierten) Einheit möglich.

Wir werden zunächst Modelle mit Warteräumen aufbauen und erst später Warteschlangen einführen. Eine Warteschlange ist eine Menge, bei der die Elemente stets nach vorgegebenen Kriterien geordnet sind. Diese ständig bestehende Ordnung erleichtert die Selektion von Elementen aus einer Menge.

Um mit der Vorstellung einer Warteschlange arbeiten zu können, muß es möglich sein,

- die Zugehörigkeit einer beweglichen Einheit zu einer Warteschlange eindeutig festzulegen
- Ordnungskriterien für eine Warteschlange zu vereinbaren
- Teilmengen von Warteschlangen zu bilden
- die Anzahl von Elementen einer Warteschlange festzustellen
- auf einzelne Elemente aus einer Warteschlange über ihre Position selektiv zuzugreifen.

Eine alternative Sicht der Warteschlange ist es, nur das Einreihen nach angegebenen Ordnungskriterien vorzunehmen und die dadurch geschaffene Ordnung aufrecht zu erhalten. Ändert sich ein Attibut eines Elements aus dieser Warteschlange, so führt das nicht zu einer Änderung der bestehenden Ordnung. Für diese Art von Warteschlange wäre es denkbar die Einreihung, nach einem an die Warteschlange gebundenen Kriterium vorzunehmen oder beim Einreihen individuell die Position zu bestimmen, an die das Element gebracht werden soll. Beide Möglichkeiten können auch nebeneinander zur Anwendung kommen.

Eine besondere Beachtung verdient die Selektion von Elementen aus einer Warteschlange. Die Kriterien können sehr vielfältig sein und dennoch sollte sich die Selektion verständlich in der Modellbeschreibungssprache formulieren lassen. Folgende Beispiele sind typisch für die Selektion aus Warteschlangen bzw. Warteräumen:

- Zugriff auf den ältesten Auftrag in der Warteschlange (FIFO)
- Zugriff auf den jüngsten Auftrag in der Warteschlange (LIFO)
- Zugriff auf den ältesten Auftrag, der eine bestimmte Eigenschaft erfüllt
  (z.B. im Arbeitsspeicher noch Platz findet)
- Zugriff auf die fünf ältesten Teile in der Warteschlange
  (z.B. Sammelstation zum Verpacken)
- Zugriff auf denjenigen Auftrag der als erster fertig wird
  (z.B. Mehrfachbedienstation)
- Zugriff auf bis zu drei Aufträge mit höchster Priorität, die am längsten gewartet haben
  (z.B. Zusammenstellung einer Fahrtroute)

Wir wollen nun Zug um Zug die für die Formulierung von Warteschlangen- und Transportmodellen geforderte Funktionalität herstellen und dazu neue Sprachkonstrukte einführen.

## 3.3 Einführung von Elementen (mobilen Komponenten)

In Warteschlangenmodellen bewegen sich Einheiten, im folgenden Elemente genannt, die sich durch eine Zusammenfassung von Attributen auszeichnen. Um die Zusammengehörigkeit dieser Attribute deutlich werden zu lassen, führen wir das Element als Datenstruktur (Verbund von Attributen) ein.

Im Vorgriff auf die spätere Verwendung bezeichnen wir Elemente auch als mobile Elemente oder mobile Komponenten.

```
mobile_component ::=   MOBILE COMPONENT  identifier
                       definition_part
                     [ declaration_part ]
                       END OF  identifier
```

Das Element ist wie eine Basiskomponente zu formulieren, mit dem Unterschied, daß es keine eigene Dynamik besitzt.

Beispiel: Die Elemente eines Warteschlangenmodells haben die Attribute `Ort`, `Pos`, `Farbe` und `Laenge`. Das Attribut `Ort` bezeichnet die Warteschlange, in der sich das Element gerade befindet, das Attribut `Pos` die Stellung innerhalb der Warteschlange.

```
MOBILE COMPONENT  Job

DEFINITIONS
   VALUE SET  Queues: ('Q1', 'Q2', 'Q3')
   VALUE SET  Color:  ('red', 'green', 'blue')

DECLARATIONS
   STATE VARIABLES
   CONSTANT   Farbe (Color) :=  'red' ,
              Laenge (REAL) :=   5

   DISCRETE   Ort (Queues)  :=  'Q1' ,
              Pos (INT)     :=   1

END OF  Job
```

Die für die Verwendung in Basiskomponenten eingeführten Konstanten sind Größen, die während des gesamten Simulationslaufs auf einem konstanten Wert bleiben. In den meisten Anwendungsfällen reicht die Lebensdauer mobiler Komponenten nicht über den gesamten Simulationslauf, sondern nur für den Weg, den sie von einer Quelle (d.h. ihrer Erzeugung) bis zu einer Senke (d.h. ihrer Vernichtung) durchlaufen. Es ist daher dafür zu sorgen, daß Konstanten von mobilen Komponenten bei ihrer Erzeugung gesetzt werden können. Darauf wird allerdings erst im Kapitel 4 eingegangen.

Der Zugriff auf ein Attribut eines Elements erfolgt zweistufig über Elementnamen und Attributnamen, getrennt durch einen Punkt.

```
attribute ::=   identifier
              | identifier '.' identifier
```

Eine Schachtelung von Elementen (Element auf Element) ist zunächst nicht erforderlich und wird später in einem anderen Zusammenhang eingeführt.

Die Syntax von Definitionsgleichungen ist schließlich noch dahingehend abzuändern, daß auch Attributen von Elementen Werte zugewiesen werden können.

```
assignment ::=  attribute [id_specificator]  ':='  conditional_expression ';'
```

Beispiel:

```
Job.Ort^ :=  'Q2'  WHEN  X > 0  ELSE
             'Q3'                END
```

## 3.4 Einführung von Feldern

In einem Warteschlangensystem bewegen sich meist sehr viele Elemente. Um nicht für jedes einzelne Element eine Beschreibung abgeben zu müssen, führen wir indizierbare Elemente, d.h. Felder ein.

Später werden sich zwar Felder für die Beschreibung von Warteschlangenmodellen als nicht unbedingt erforderlich herausstellen, die damit verbundenen sprachlichen Konstrukte, insbesondere der Allquantor, lassen sich anhand von Feldern jedoch besser einführen. Darüber hinaus sollte wegen einer Vielzahl anderer Anwendungen auf Felder ohnehin nicht verzichtet werden.

Jede Modellvariable bzw. jedes mobile Element sowie jede Modellkomponente kann als Feld (Reihung) mit einer oder mehreren Dimensionen ausgebildet sein. Ein einzelnes Feldelement wird durch die Angabe von Indizes (je einer pro Dimension) identifiziert.

Die Indices umfassen einen zusammenhängenden Bereich ganzer Zahlen, der bei der Deklaration angegeben wird.

### 3.4.1 Deklaration

Jeder Bezeichner für eine Variable oder eine Komponente (auch ein Element) kann als Feld vereinbart werden.

```
dimensioned_identifier ::=   [ ARRAY  dimension_list ]  identifier

dimension_list         ::=   '{' index_range '}'  { '{' index_range '}' }

index_range            ::=   constant_integer '..' constant_integer

constant_integer       ::=   operand
```

Einem Bezeichner, der als Feld deklariert werden soll, wird das Schlüsselwort `ARRAY` und eine Dimensionierungsliste vorangestellt. Für jede Dimension wird in geschweiften Klammern der Indexbereich (Indexmenge) angegeben.

Die Feldgrenzen sind durch konstante, ganze Zahlen festzulegen. Die Verwendung von Dimensionierungskonstanten ist ebenso erlaubt wie Operanden (z.B. geklammerte Ausdrücke, siehe [Esch 90]), die sich aus konstanten, ganzen Zahlenwerte zusammensetzen und sich bereits zur Compile-Zeit berechnen lassen.

Beispiele:

```
ARRAY {1..100} X                    ARRAY {1..10} {1..5} Y
ARRAY {-5..+5} X                    ARRAY {0.. 9} {0..4} Y

ARRAY {1..XDIM} X                   ARRAY {1..(XDIM-1)} {1..(2*XDIM)} Y

ARRAY {(XDIM-5)..(XDIM+5)} XM
```

### 3.4.2 Verwendung von indizierten Variablen in Definitionsgleichungen

Durch die Angabe eines konkreten Index läßt sich jedes einzelne Feldelement - auch von mobilen Komponenten - ansprechen.

```
indexed_attribute    ::=   indexed_identifier
                         | indexed_identifier '.' indexed_identifier

indexed_identifier   ::=   identifier  [ index_list ]

index_list           ::=   index { index }

index                ::=   '[' index_expression ']'

index_expression     ::=   expression
```

Hinter dem Feldnamen steht die Indexliste, für jede Dimension ein Indexausdruck in eckigen Klammern.

Beispiele:

```
X [2]                  Y [2][5]
X [2*k+1]              Y [k+1][k]
```

Der Indexausdruck muß einen ganzzahligen Wert liefern, der im Indexbereich der jeweiligen Dimension liegt.

Die Werte von Feldelementen werden - wie gehabt - durch Definitionsgleichungen festgelegt. Hierzu ist die Syntax der Wertzuweisung entsprechend abzuändern.

```
assignment ::=  indexed_attribute [tr_spec]  ':='  conditional_expression ';'
```

Um die Eindeutigkeit und Vollständigkeit sicherzustellen, sind folgende Bedingungen zu erfüllen:

- Jedes einzelne Feldelement muß eindeutig definiert sein, d.h. für jeden Index darf nicht mehr als Modellgleichung gültig sein.
- Damit keine überflüssigen Variablen deklariert sind, sollte jedes einzelne Feldelement zumindest in Teilen des Zustandsraums durch eine Modellgleichung definiert sein.
- Abhängige Variable müssen vollständig, d.h. für jeden Index in allen Teilen des Zustandsraums durch eine gültige Modellgleichung definiert sein.

Um diese Bedingungen wie bisher vom Compiler überprüfen zu können, ist es erforderlich, die Indices der definierten Variablen zu kennen. Inwieweit dies möglich ist, wird an späterer Stelle erörtert.

Zunächst jedoch wollen wir den Allquantor einführen, um indizierte Variablen wirkungsvoll einsetzen zu können.

### 3.4.3 Formulierung von Definitionsgleichungen mit Allquantoren

Um formulieren zu können, daß Definitionsgleichungen nicht nur für einen einzigen Variablenindex sondern für eine ganze Menge von Indices gelten, führen wir Allquantoren ein.

Da die Aussageformen, die mit einem Allquantor zusammengefaßt werden, stets wahr sind, liefert der hier eingeführte Allquantor keinen logischen Wert zurück. Er unterscheidet sich insofern vom Allquantor wie er in der Logik Verwendung findet und kann daher auch nicht als Bestandteil von logischen Ausdrücken gebraucht werden. Um dies deutlich zu machen, verwenden wir anstelle des Schlüsselwortes `FORALL` das Schlüsselwort `FOREACH`.

```
foreach_statement  ::=   FOREACH  iterator { ',' iterator }
                         LET
                              statement_sequence
                         END

iterator           ::=   identifier  IN  index_range  [ '|' condition ]
```

Mit dem FOREACH-Konstrukt wird eine gebundene Variable vereinbart, deren Wert über einem angegebenen Indexbereich variiert. Durch Angabe einer Bedingung kann die Indexmenge eingeschränkt werden. Die gebundene Variable besitzt Gültigkeit innerhalb einer eventuell angegebenen Bedingung sowie im Rumpf der Anweisung zwischen LET und END.

Beispiele:

```
FOREACH  i  IN  2..(XDIM-1)
LET
    DIFFERENTIAL EQUATION
        X[i]' :=  2 * X[i] - X[i-1] - X[i+1];
    END
END
```

Die Ableitung der Feldelemente `X[i]` ermittelt sich aus den Werten des Feldelements und dessen Nachbarn.

```
FOREACH  i IN 1..M,  j IN 1..N
LET
    Y[i][j] :=  A[i][j] + B[i][j];
END
```

Die Werte der Feldelemente `Y[i][j]` ergibt sich zu jedem Zeitpunkt aus der Summe der Feldelemente `A[i][j]` und `B[i][j]`.

```
FOREACH  i  IN  1..10
LET
    Z[i]^ :=  TRUE  WHEN  X[i] > 0  AND  Z[i] = FALSE  END
END
```

Sobald für irgendein `i` die Variable `X[i]` positiv wird, wird `Z[i]` auf `TRUE` gesetzt und nicht mehr zurückgenommen.

Wenngleich der Allquantor sehr an das Schleifenkonstukt einer prozeduralen Programmiersprache erinnert und selbstverständlich auch durch ein solches implementiert wird, so ergeben sich durch die semantischen Anforderungen einer Spezifikationssprache doch eine Reihe von Einschränkungen. Wir betrachten die Anforderungen an die Definition indizierter Variablen zunächst allgemein und später im Zusammenhang mit dem Allquantor.

### 3.4.4 Semantisch korrekter Gebrauch von indizierten Variablen in Definitionsgleichungen

Die Überschreitung des Indexbereiches ist für gewöhnlich ein Problem, das erst zur Laufzeit erkannt werden kann. Für die Überprüfung der Eindeutigkeit und Vollständigkeit ist jedoch die Kenntnis der einzelnen Indices erforderlich.

Deshalb wollen wir darüber nachdenken, unter welchen Einschränkungen eine Indexkontrolle durch den Übersetzer möglich wird und ob diese Einschränkungen die Verwendung von Feldern nicht zu sehr einengen.

Solange nur konstante Indices bzw. Indexausdrücke zulässig sind, hat man die gleichen Verhältnisse wie bei einzelnen Variablen.

Beispiel:

```
x[1] :=  y[1] + z[1];
x[2] :=  y[2] + z[2];
```

Allerdings ergibt die Verwendung von Feldern hierbei keine Vorteile. Interessant wird erst der Fall, daß ein Indexausdruck eine Variable enthält.

**Indexprüfung auf der rechten Gleichungsseite**

Wir wollen dabei zunächst den Fall betrachten, daß ein variabler Index auf der rechten Gleichungsseite auftaucht.

Beispiel:

```
ARRAY {0..4}  X (REAL)

IF  T >= TNext
LET
    Z^ :=  X[k];
    k^ :=  (k+1) MOD 5;
    TNext^ :=  T + 10;
END
```

Alle 10 Zeiteinheiten wird einer von 5 Eingängen `X[0] .. X[4]` abgetastet und der Abtastwert in der Variablen `Z` festgehalten.

Wie man relativ schnell erkennt, gibt es keine Möglichkeit, bereits durch den Compiler festzustellen, ob die Variable `k` stets im Bereich zwischen 0 und 4 liegt. Dies wäre nur durch eine symbolische Analyse zu leisten. Da bislang auch keine syntaktische Notation gefunden wurde, die eine Prüfung unterstützt, ist eine Kontrolle des Indexbereichs zur Compilezeit im allgemeinen nicht möglich.

Handelt es sich bei dem Index allerdings um die gebundene Variable eines FOREACH-Konstrukts, wie das in der Praxis meist der Fall ist, so ist selbstverständlich bekannt, innerhalb welcher Grenzen die Werte dieser Variablen liegen. Es spricht dann natürlich nichts dagegen, bei speziellen, aber häufig vorkommenden Indexausdrücken wie `i+1` oder `2*i+1`, welche die gebundene Variable durch einfache Operationen mit Konstanten verknüpfen, bereits vom Compiler eine Feldgrenzenprüfung durchführen zu lassen.

Beispiel:

```
FOREACH  i  IN  2..9
LET
     Z[i] :=  Y[i-1] + Y[i+1];
END
```

Erleichtert wird eine Feldgrenzenüberwachung bei einer Beschränkung auf monotone Operationen wie `+`, `-` oder `*`, da in diesen Fällen nur für den kleinsten und den größten Index der gebundenen Variable ein Test durchgeführt werden muß.

### Indexprüfung auf der linken Gleichungsseite

Erscheint der variable Index auf der linken Gleichungsseite, d.h. bei der zu definierenden Variablen, ergibt sich eine ähnliche Situation.

Beispiel:

```
Z[k]^ :=  TRUE  WHEN  k > 0  AND  Z[k] = FALSE  END
```

Die Variable `k` habe für gewöhnlich den Wert null. Sobald `k` einen positiven Wert annimmt, wird `Z[k]` auf `TRUE` gesetzt und nicht mehr zurückgenommen, auch wenn `k` wieder null wird oder einen anderen Wert annimmt. In dem Feld `Z` wird demnach festgehalten, welche Werte die Variable `k` im Lauf ihrer Geschichte angenommen hat.

Eine Verletzung der Feldgrenzen läßt sich nur sicher vermeiden, wenn man gewährleisten kann, daß die Variable `k` aus der Indexmenge des Feldes `z` entnommen ist. Auch dies ist wiederum nur möglich, wenn es sich um die gebundene Variable eines FOREACH-Konstrukts handelt.

### Prüfung auf Vollständigkeit

Abhängige Variablen müssen vollständig, d.h. für jeden einzelnen Index definiert sein. Für eine Überprüfung zur Compilezeit ist es deshalb erforderlich, daß die Indices berechenbar sind. Definiert man ein Feld von abhängigen Variablen, dann darf der Indexausdruck folglich keine anderen Variablen als die gebundene Variable eines Allquantors enthalten. Die Praxis zeigt (glücklicherweise), daß kein Bedarf dafür besteht, andere Variablen im Index zu verwenden.

Beispiel:

```
ARRAY {1..N} X(REAL), ARRAY {1..N} Y(REAL)

Y[k] :=  0;
FOREACH  i  IN  2..N
LET
    Y[i] :=  2 * X[i];
END
```

Da man nicht davon ausgehen kann, daß die Variable `k` stets den Wert 1 annimmt, muß man eine unvollständige (und wahrscheinlich auch nicht eindeutige) Definition annehmen.

### Prüfung auf Eindeutigkeit

Die Eindeutigkeit hingegen läßt sich durch eine weniger weit gehende Forderung sichern: Enthält der Indexausdruck einer zu definierenden Variablen eine andere als eine gebundene Variable, verbietet man lediglich, daß für irgendein Feldelement dieser Variablen eine weitere, gleichzeitig gültige Definitionsgleichung in der Modellbeschreibung erscheint.

Unterstellt man nämlich, daß der variable Index jeden möglichen Wert annehmen kann, dann kommt es wenigstens bei einem Wert zum Konflikt.

Beispiel:

```
ARRAY {0..4}  Z (REAL)

IF  T >= TNext
LET
    Z[0]^  :=  0  WHEN  k = 0  AND  x < 0  END
    Z[k]^  :=  x;
    k^     :=  (k+1) MOD 5;
    TNext^ :=  T + 10;
END
```

Alle 10 Zeiteinheiten wird die Variable `x` abgetastet und die letzten 5 Werte in den Feldelementen der Zustandsvariable `Z` zyklisch abgespeichert. Dabei soll die Zustandsvariable `Z[0]` den Wert null annehmen, wenn `x` kleiner als 0 ist. Für `k=0` und `x<0` ist aus der angegebenen Modellbeschreibung jedoch nicht eindeutig ersichtlich, welchen Wert `Z[0]^` annehmen soll.

Eine Lösung durch die oben genannte Einschränkung ist tragbar, da sich solche Probleme durch eine alternative syntaktische Formulierung aus der Welt schaffen lassen. Dies hat den Vorteil, daß der mögliche Konflikt durch die Fallunterscheidung deutlich zu Tage tritt.

Beispiel:

```
IF  T >= TNext
LET
    Z[k]^  :=  0  WHEN  k = 0  AND  x < 0     ELSE
               x                              END
    k^     :=  (k+1) MOD 5;
    TNext^ :=  T + 10;
END
```

Gemäß obiger Forderung enthält die Modellbeschreibung nur eine einzige Definitionsgleichung für das Feld `z`. Es ist zwar nicht möglich, die Einhaltung der Feldgrenzen zu gewährleisten, die Eindeutigkeit ist jedoch sichergestellt.

**Zusammenfassung:**

Besitzt eine zu definierende Variable einen Indexausdruck, der eine Variable enthält, die keine gebundene Variable ist, dann ist

- keine Überprüfung der Feldgrenzen durch den Compiler möglich
- keine Vollständigkeit gewährleistet
- Eindeutigkeit nur gegeben, wenn nicht gleichzeitig eine zweite Definitionsgleichung für irgendein Feldelement dieser Variablen vorhanden ist.

Eine Überprüfung der Feldgrenzen durch den Compiler ist nur möglich, wenn

- ein Indexausdruck außer einer gebundenen Variablen keine weiteren Variablen enthält.

Vereinfacht wird die Kontrolle der Feldgrenzen, wenn ein Indexausdruck

- nur eine einzige gebundene Variable enthält
- nur solche Operatoren enthält, die eine monotone Funktion erzeugen.

Die hier behandelten Einschränkungen sind allgemeiner Natur. Wir wollen nun untersuchen, welche Konsequenzen dies für die Verwendung von Definitionsgleichungen innerhalb eines FOREACH-Konstrukts nach sich zieht.

### 3.4.5 Semantisch korrekter Gebrauch von Allquantoren

Zunächst läßt sich feststellen, daß es keinen Sinn ergibt, im FOREACH-Statement Modellvariablen zu definieren, ohne die gebundenen Variablen im Index zu verwenden.

Beispiel:

```
FOREACH  i  IN  1..10
LET
    A    :=  10;           # Ausfuehrung ineffizient
    B    :=  Y[i];         # Belegung von B widerspruechlich
    X[i] :=  Y[i];         # korrekte Verwendung
END
```

Eine solche Formulierung ist nicht nur in der Ausführung ineffizient, sie wird auch widersprüchlich, wenn der zugewiesene Ausdruck den iterierten Index enthält. Es ist daher zu fordern, daß Variablen, die definiert werden, alle gebundenen Variablen in ihren Indices verwenden.

Das folgende Beispiel zeigt, daß diese Forderung jedoch noch nicht ganz ausreichend ist.

Beispiel:

```
FOREACH  i  IN  1..10,  j  IN  1..5
LET
    U[i+j]         :=  Z[i][j];      # Indices treten mehrfach auf
    X[i][j MOD 2]  :=  Z[i][j];      # Indices treten mehrfach auf
    Y[i][j]        :=  Z[i][j];      # korrekte Verwendung
END
```

Es zeigt sich, daß offensichtich in einer Reihe von Fällen Indices mehrfach auftreten können. Um die Eindeutigkeit zu sichern, ist deshalb durch geeignete Einschränkungen Sorge zu tragen, daß jedes Feldelement nur ein einziges Mal definiert wird.

Wir verlangen daher, daß jeder Index nur eine einzige gebundene Variable enthalten darf und der Indexausdruck eine monotone Funktion sein muß. Wie bereits erwähnt, erleichtert die letzte Forderung auch die Überwachung der Feldgrenzen und ist beispielsweise dann erfüllt, wenn nur die Operatoren `+`, `-` und `*` zugelassen werden.

Unter der Voraussetzung, daß nicht eine Variable durch zwei oder mehrere Definitionsgleichungen beschrieben wird, ist die Eindeutigkeit hergestellt.

Bereits im letzten Abschnitt wurde festgestellt, daß pro Bezeichner nur eine einzige Definitionsgleichung angegeben werden darf, wenn der Index eine Variable enthält. Es könnte sonst zu Überschneidungen der Indices und damit zu einer Verletzung der Eindeutigkeit kommen.

Auch mit gebundenen Variablen innerhalb eines FOREACH-Konstrukts können Schwierigkeiten dieser Art entstehen.

Beispiel:

```
FOREACH  i  IN  1..5
LET
    X[2*i]   :=  1;         # Belegung bei geradem Index

    X[2*i-1] := -1;         # Belegung bei ungeradem Index
END
```

In diesem Beispiel kommt es noch zu keinem Widerspruch, während es im folgenden Beispiel zu einer Überschneidung von Indices kommen kann.

Beispiel:

```
FOREACH  i  IN  1..9         #  Das Feld X soll durch eine Folge
LET                          #  von Ereignissen aufsteigend
    IF  X[i] > X[i+1]        #  sortiert werden.
    LET
        X[i]^   :=  X[i+1];

        X[i+1]^ :=  X[i];
    END
END
```

Beim Sortieren der Teilfolge `X[1] = 3` `X[2] = 2` `X[3] = 1` wird der Variablen `X[2]^` sowohl der Wert `3` (für `i=1`) als auch der Wert `1` (für `i=2`) zugeordnet.

Diese möglichen Überschneidungen lassen sich nur feststellen, wenn der Compiler eine Liste aller auftretenden Indices anlegt. Dieser Aufwand läßt sich vermeiden, wenn man innerhalb eines FOREACH-Konstrukts nur eine Definitionsgleichung pro Variable zuläßt. Noch einfacher wird eine Prüfung auf semantische Korrektheit, wenn als Index nur die gebundene Variable alleine auftreten darf.

In beiden Fällen ist daher eine alternative Syntax zu verwenden.

Beispiel:

```
FOREACH  i  IN  1..10
LET
    IF  MOD (i,2) = 0
    LET
        X[i] :=  1;         # Belegung bei geradem Index
    END
    ELSE
    LET
        X[i] := -1;         # Belegung bei ungeradem Index
    END
END
```

Im zweiten Beispiel dehnt sich die Modellbeschreibung etwas in die Länge, was aber keine Folge der getroffenen Einschränkungen ist, sondern sich daraus ergibt, daß die kleinsten und größten Indices gesondert berücksichtigt werden müssen.

Beispiel:

```
IF  X[1] > X[2]
LET
    X[1]^ :=  X[2];
    X[2]^ :=  X[1];
END
ELSIF  X[2] > X[3]
LET
    X[2]^ := X[3];
END

FOREACH  i  IN  3..(N-1)
LET
    IF     X[i] < X[i-1]
    LET
           IF  X[i-1] > X[i-2]
           LET
               X[i]^ :=  X[i-1];
           END
    END
    ELSIF  X[i] > X[i+1]
    LET
           X[i]^ := X[i+1];
    END
END

IF  X[N-1] > X[N]  AND  X[N-1] >= X[N-2]
LET
    X[N]^ := X[N-1];
END
```

Das Vertauschen der Werte mit dem linken Nachbarn hat Vorrang, wird aber nur durchgeführt, wenn nicht der linke Nachbarn möglicherweise selbst mit seinem linken Nachbarn tauscht.

Die nun gefundene Beschreibung zum Sortieren des Zustandsvektors `X` ist semantisch korrekt und bei genauerem Hinsehen stellt man fest, daß sie auch als Algorithmus für einen Parallelrechner dienen kann. Dies ist nicht weiter verwunderlich, wenn man bedenkt, daß Ereignisse gleichzeitig ausführbare Zuweisungen sind.

Weiterhin stellt man fest, daß man sich beim Abfassen der Beschreibung quasi auf den Standpunkt einer beliebigen Variablen stellt und deren Veränderung beschreibt. Diese Vorgehensweise ist typisch für die Beschreibung des Verhaltens von Modellvariablen.

Die getroffenen Einschränkungen mögen sehr restrikiv erscheinen, stellen aber keine prinzipiellen Beschränkungen dar, was die Formulierbarkeit von Modellen anbelangt. Insbesondere erleichtern sie auch die Prüfung auf Vollständigkeit, da bekannt ist, für welche Feldindices eine Variable definiert ist.

Wenn abhängige Variablen zu definieren sind, ist als weitere Einschränkung lediglich zu fordern, daß keine Bedingung den Indexbereich eines FOREACH-Konstrukts einschränken darf.

Beispiel:

```
FOREACH  i  IN  1..10  |  Z[i] > 0
LET
    X[i] :=  Z[i];
END
```

Die abhängigen Variablen `X[i]` sind nur definiert, falls `Z[i] > 0` ist.

Eine korrekte Formulierung könnte folgendermaßen aussehen.

Beispiel:

```
FOREACH  i  IN  1..10
LET
    IF    Z[i] > 0   LET   X[i] :=  Z[i];   END
    ELSE             LET   X[i] :=  0;      END
END
```

Die abhängigen Variablen `X[i]` sind im gesamten Zustandsraum definiert, da der ELSE-Teil nicht fehlt.

**Zusammenfassung:**

Das Sicherstellen der Eindeutigkeit erfordert beim Gebrauch des Allquantors, daß

- für jeden Feldnamen nur eine einzige Definitionsgleichung angegeben ist
- jeder Index der zu definierenden Variablen nur aus einer gebundenen Variablen besteht
- alle gebundenen Variablen als Indices der zu definierenden Variablen verwendet werden

Die Gewährleistung der Vollständigkeit erfordert darüber hinaus, daß

- das FOREACH-Konstrukt keine Bedingung enthält

Eine Überprüfung der Feldgrenzen ist unter diesen Bedingungen stets möglich.

### 3.4.6 Impliziter Allquantor

Um die Vollständigkeit und Eindeutigkeit zu sichern, haben wir zahlreiche Einschränkungen getroffen. Als besonders vorteilhaft stellte sich heraus, den Indexausdruck der zu definierenden Variablen auf die gebundene Variable zu beschränken.

Es bietet sich daher alternativ eine kompaktere Notation für den Allquantor an, die sich lediglich auf eine einzige Definitionsgleichung bezieht.

Dazu erlaubt man, daß Indices der zu definierenden Variablen eine Indexmenge anstelle eines einzelnen Wertes enthalten dürfen.

```
index ::=   '[' index_expression  ']'
          | '{' implicit_indexset '}'

implicit_indexset ::=   index_range
                      | identifier  [ IN index_range ]  [ '|' condition ]
```

Diese Indexmenge (in geschweiften Klammern) umfaßt einen Indexbereich, der durch eine Bedingung eingeschränkt werden kann, und kann mit einer gebundenen Variablen versehen werden. Wird kein Indexbereich angegeben, dann umfaßt die gebundene Variable alle möglichen Indices dieser Dimension. Die gebundene Variable wird implizit deklariert und besitzt nur innerhalb der Definitionsgleichung Gültigkeit.

Bei der Definition von abhängigen Variablen darf keine den Indexbereich einschränkende Bedingung verwendet werden.

Beispiel:
```
X {i}  :=  2 * Y[i];
```

Beispiel:
```
X [1]           :=  Y[1];
X {i IN 2..10}  :=  2 * Y[i];
```

Beispiel:
```
Z {i}^ :=  'High'  WHEN  X[i] >  10  ELSE
           'Low'   WHEN  X[i] < -10  END
```

Beispiel:
```
Z {i | i < 0}^ :=  -Z[i]  WHEN  X[i] < Y[i]  AND  Z[i] < 0 END
```

Der implizite Allquantor läßt dem Anwender etwas weniger Freiheiten, ermöglicht aber eine griffige und kompakte Formulierung sowie einen sicheren Gebrauch, ohne daß man etwas über semantische Einschränkungen wissen müßte.

## 3.5 Einführung von Indexmengen

Im ersten Ansatz formulieren wir Warteschlangenmodelle, indem wir durch Mengenbildung feststellen, welche mobilen Elemente sich auf demselben Aufenthaltsort befinden und welche Elemente bzw. wieviele bestimmte Eigenschaften besitzen.

### 3.5.1 Bildung von nicht geordneten Indexmengen

Indexmengen werden für gewöhnlich durch Aufzählung einzelner Indices bzw. Indexbereiche oder durch die Angabe von Eigenschaften beschrieben. Für unsere Zwecke kann die Bildung von Mengen sehr einfach gehalten werden.

Mit Rücksicht auf die Lesbarkeit dürfen Eigenschaftsmengen daher lediglich durch Einschränkung eines einzigen Indexbereichs gebildet werden. Geschachtelte Konstruktionen, bei denen eine Eigenschaftsmenge Teilmenge einer anderen Eigenschaftsmenge ist, werden nicht ermöglicht. Für die Modellbildung erweist sich die vorgeschlagene Mengendefinition als ausreichend.

```
index_set ::=       cardinal_set
                  | property_set

cardinal_set ::=    '{' index_range '}'
                  | '{' constant_integer '}'

property_set ::=    '{' identifier  IN  index_range  [ '|' condition ]  '}'

condition    ::=    expression
```

Für die Bildung einer Eigenschaftsmenge wird ein Bezeichner für einen Index angegeben, der implizit deklariert wird und nur innerhalb der Mengenklammern Gültigkeit besitzt.

Beispiele:

```
{ 10..20 }                          #  cardinal_set
{ 5 }

{ i  IN  1..10 }                    #  property_set
{ i  IN  1..10  |  X[i] = 0 }
{ i  IN  1..N   |  Job[i].Ort = 'Q1' }
```

Wir wollen auch einige einstellige Operatoren einführen, die eine Indexmenge als Argument besitzen.

`CARD  index_set` liefert als ganze Zahl die Kardinalität, also die Anzahl der Mengenelemente

`EMPTY index_set` liefert einen logischen Wert:
`TRUE` wenn die Menge leer ist
`FALSE` sonst

`ALL index_set` liefert für Eigenschaftsmengen einen logischen Wert:
- `TRUE` wenn die Menge die gesamte Grundmenge umfaßt
- `FALSE` sonst

`ANY index_set` liefert für Eigenschaftsmengen einen logischen Wert:
- `TRUE` wenn die Menge wenigstens ein Element enthält
- `FALSE` sonst

Beispiel: `CARD { i IN 1..N | Job[i].Ort = 'Server' }`

Das Schlüsselwort IN soll uns auch als zweistelliger Operator dienen. Die Operation

```
identifier IN indexset
```

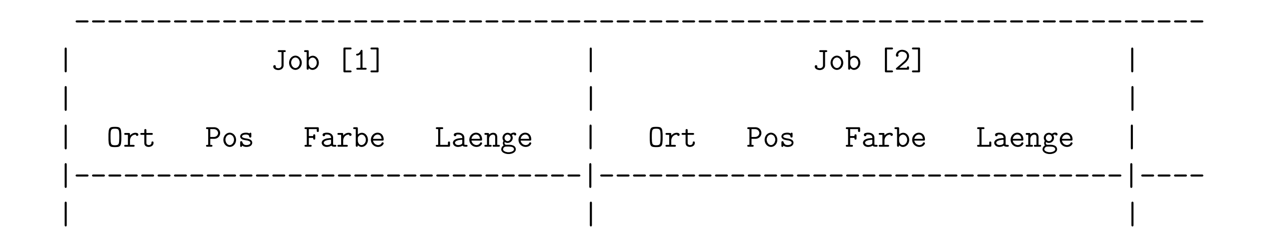
liefert einen logischen Wert:
- `TRUE` wenn die Konstante in der Indexmenge enthalten ist
- `FALSE` sonst

Beispiel: `IF j IN { i IN 1..N | X[i] = 0 } LET ... END`

### 3.5.2 Problematik beim Transport zwischen Warteräumen

Mit den besprochenen Spracherweiterungen sind wir mittlerweile in der Lage, das eingangs im Unterabschnitt 'Motivation' dargestellte Beispiel zu verstehen.

Die Elemente des Warteschlangensystems legen wir als Feld an.


```
 -----------------------------------------------------------------------
|            Job [1]            |             Job [2]             |
|                               |                                 |
| Ort   Pos   Farbe   Laenge    |  Ort   Pos   Farbe   Laenge     |
|-------------------------------|---------------------------------|----
|                               |                                 |
```

Wir modellieren das Verhalten jedes einzelnen Elements. Hierzu indizieren wir das Element und erweitern die Aussage mit Hilfe des eingeführten FOREACH-Konstrukts für alle Elemente.

```
ARRAY {1..N}  Job

FOREACH  i  IN  1..N
LET
    IF  Job[i].Ort = 'Q'  AND  EMPTY {j IN 1..N | Job[j].Ort = 'S'}
    LET
        IF  Job[i].Pos = 1
        LET
            Job[i].Ort^ :=  'S';
        ELSE
        LET
            Job[i].Pos^ :=  Job[i].Pos - 1;
        END
    END
END
```

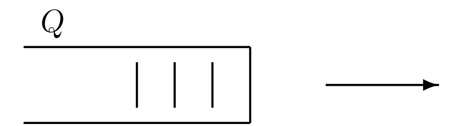
Aufgrund des Attributes `Ort` werden alle mobilen Elemente an einem Aufenthaltsort zu einer Menge zusammengefaßt. Das Attribut `Pos` soll eine Ordnung innerhalb dieser Menge herstellen.

Beim Hinzufügen eines Elements zu einer Menge ist daher seine neue Position innerhalb dieser Menge zu bestimmen. Wird es vor der bisher letzten Position eingereiht, müssen die dahinterliegenden Elemente um eine Position nach hinten verschoben werden. Im Beispiel ist dies nicht erforderlich, da nur ein Element im Server Platz findet.

Bei der Entnahme eines Elements aus einer Menge rücken diejenigen Elemente, die hinter dem entnommenen Element eingereiht waren, um eine Position nach vorne.

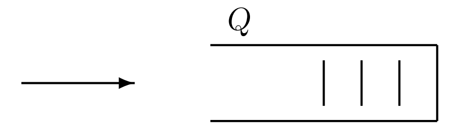
Dieses Verfahren, für jedes Element eine Position festzulegen, ist nicht nur umständlich. Wie die folgenden Beispiele deutlich werden lassen, ist das Hinzufügen und Entnehmen von Elementen in etwas schwierigeren (aber immer noch typischen) Fällen nicht formulierbar.

Beispiel für eine Entnahme aus einer Warteschlange:




Bei Eintritt eines bestimmten Ereignisses, sollen die zwei längsten Elemente (Attribut `Laenge`), die kleiner oder gleich einer gegebenen maximalen Länge `LMax` sind, aus der Warteschlange entnommen und an einen anderen Ort gebracht werden. Falls zwei Elemente gleich lang sind, soll dasjenige ausgewählt werden, das die Warteschlange zuerst betreten hat.

Beispiel für das Einreihen in eine Warteschlange:




> Ankommende Elemente sind in der Warteschlange ihrer Länge nach zu ordnen, die längsten Elemente vorne, die kürzesten hinten. Sind zwei Elemente gleich lang, wird das neu ankommende hinten angestellt.

Der Versuch, Problemstellungen dieser Art zu modellieren, zeigt: Das Festlegen des Positionsattributs ist (mit den bisherigen Mitteln) entweder überhaupt nicht möglich, oder nur äußerst umständlich und aufwendig zu formulieren.

Deshalb werden wir für die Selektion von Elementen einen ganz anderen Weg beschreiten: Wir schaffen uns sprachliche Mittel, mit deren Hilfe wir eine Menge ordnen und den Zugriff auf Elemente geordneter Mengen formulieren können. Dadurch wird das Positionsattribut überflüssig und auch das Einreihen in eine Warteschlange ist nicht mehr erforderlich.

### 3.5.3 Einführung geordneter Mengen

Die Ordnung einer Menge bzw. Indexmenge wird hergestellt, indem nach monoton aufsteigenden oder abfallenden Werten einer Funktion geordnet wird.

```
ordered_property_set ::=  '{'  identifier  IN  index_range
                               [ '|'  condition ]
                               [ '||' criteria_list ]  '}'

criteria_list        ::=  ordering_criterion { ',' ordering_criterion }

ordering_criterion   ::=  INC   expression
                        | DEC   expression
                        | TRUE  expression
                        | FALSE expression
```

Ein Ordnungskriterium enthält einen numerischen oder logischen Ausdruck. Ist der Ausdruck numerisch, dann soll sein Wert mit jedem Element in der Warteschlange entweder zunehmen (INC) oder abnehmen (DEC). Ist der Ausdruck logisch, dann werden die Elemente mit denjenigen Indices vorangestellt, bei denen der Ausdruck den angegebenen logischen Wert annimmt.

Der Ausdruck des Ordnungskriteriums muß selbstverstendlich den variierten Index beinhalten. Sollen durch das Kriterium Elemente eines Warteraums geordnet werden, dann wird der Ausdruck in der Regel ein Attribut dieser Elemente enthalten.

Liefert der Ausdruck für mehrere Elemente den gleichen Ordnungswert, dann ist ihre Reihenfolge innerhalb der Menge nicht festgelegt. Durch Hinzufügen eines weiteren Ordnungskriteriums kann bei Gleichheit des Ordnungswertes über die Reihenfolge entschieden werden.

Beispiel:

```
{ i IN 1..100  |  Job[i].Ort = 'Q1'  ||  DEC Job[i].Priority,
                                          INC Job[i].TEntry    }
```

Die Menge von Jobs auf der Warteschlange `Q1` wird nach fallenden Prioritäten geordnet. Besitzen zwei oder mehrere Jobs die gleiche Priorität, werden diese nach dem zunehmendem Eintrittszeitpunkt `TEntry` geordnet.

Erfolgte bislang eine Teilmengenbildung ausschließlich durch Angabe von Eigenschaften, so ist bei geordneten Mengen eine Teilmengenbildung auch möglich, indem man Bereiche ausgrenzt.

### 3.5.4 Beschneiden von Mengen

Das Beschneiden von Mengen, d.h. das Ausgrenzen von Teilmengen aus einer geordneten Menge, erfolgt durch Angabe desjenigen Indexbereiches, der erhalten bleiben soll.

```
cutted_set     ::=   ordered_property_set  [ ':' cardinal_set ]

cardinal_set   ::=   '{' index_range '}'
                   | '{' constant_integer '}'
```

Sind keine der angegebenen Indices in der Menge enthalten, bleibt die resultierende Menge leer.

Beispiele:

```
{ j IN 1..10  ||  INC x[j] } : {1..5}
```

Die geordnete Indexmenge wird bis auf die Positionen `1..5` abgeschnitten und enthält die Indices der 5 kleinsten Werte der Variablen `x`.

```
{ j IN 1..100 | Job[j].Laenge <= LMax  ||  DEC Job[j].Laenge } : {1}
```

Die resultierende Menge enthält nur den Index des Jobs mit dem größtem Attribut `Laenge`, der gerade noch kleiner oder gleich `LMax` ist, sofern es überhaupt einen solchen Job gibt.

### 3.5.5 Verbesserte Formulierung des Transports zwischen Warteräumen

Wir greifen die eingangs angeführten Beispiele nochmals auf und zeigen, wie sich durch die Bildung von geordneten Mengen und das Ausgrenzen von Teilmengen eine griffigere und kompaktere Beschreibung ergibt.

Beispiel:

Das erste Beispiel beschreibt den Transport des ältesten Elements aus einer Warteschlange in eine Bedieneinheit, sobald diese frei ist.

```
ARRAY {1..N}  Job

FOREACH  i  IN  1..N
LET
    IF  i IN {j IN 1..N | Job[i].Ort = 'Q' || INC Job[i].TEntry} : {1}
        AND  EMPTY {j IN Job[j] | Job[j].Ort = 'S'}
    LET
        Job[i].Ort^    := 'S';
        Job[i].TEntry^ :=  T;
    END
END
```

Das Attribut `Pos` wurde durch das Attribut `TEntry` ersetzt, das den Eintrittszeitpunkt festhält und beim Betreten eines Warteraums gesetzt wird.

Die Position wird nun indirekt über den Eintrittszeitpunkt bestimmt. Das eingangs erwähnte Problem, daß die Position nicht als Zustandsvariable im eigentlichen Sinne angesehen werden kann, hat sich dadurch erledigt.

Es fällt allerdings auf, daß die Ordnung nicht geregelt ist, wenn zwei oder mehrere Elemente den gleichen Eintrittszeitpunkt besitzen. In einem solchen Fall gehen wir davon aus, daß die Auswahl des einen oder anderen Elements weder auf das Modellverhalten noch auf die Simulationsergebnisse Einfluß hat. Wäre dies der Fall, so könnte man durch hinzuziehen weiterer Attribute die Ordnung klarstellen. Wurden bereits alle Attribute eines Elements zum Festlegen der Ordnung herangezogen, dann können nur gleiche Elemente (mit gleichen Attributen) um denselben Platz konkurrieren.

Nach einem Transport wird demnach keine Position am neuen Aufenthaltsort bestimmt, d.h. es erfolgt kein Einreihen in eine Warteschlange. Erst bei einem selektiven Zugriff wird eine Ordnung hergestellt. Ein Aufenthaltsort ist daher als Warteraum und nicht als Warteschlange anzusehen.

An einem weiteren Beispiel wird ersichtlich, daß die Entnahme aus einem Warteraum durch das Herstellen einer Ordnung auch in schwierigeren Fällen beschreibbar ist.

Beispiel:

Falls in `Q1` fünf Elemente enthalten sind, werden die zwei längsten (Attribut `Laenge`), die kleiner sind als eine maximale Länge `LMax` entnommen und nach `Q2` gebracht. Falls zwei Elemente gleich lang sind, wird dasjenige ausgewählt, das `Q1` zuerst betreten hat.

```
ARRAY {1..N}  Job

FOREACH  i  IN  1..N
LET
    IF  CARD {j IN 1..N | Job[j].Ort = 'Q1'} >= 5
    LET
        IF  i IN {j IN 1..N
                  |  Job[j].Ort = 'Q1'  AND  Job[j].Laenge <= LMax
                  || DEC Job[j].Laenge, INC Job[j].TEntry} : {1..2}
        LET
            Job[i].Ort^    :=  'Q2';
            Job[i].TEntry^ :=  T;
        END
    END
END
```

Die vorgestellten sprachlichen Mittel zur Selektion mobiler Elemente aus Warteräumen sind allgemein und elementar. Da an den Ausdruck, der die Ordnung herstellt, keine Bedingungen geknüpft sind (selbst Funktionen dürfen enthalten sein), läßt sich jede beliebige Ordnung beschreiben.

Eine Einschränkung ergibt sich nur aus der Tatsache, daß das Beschneiden von Mengen nur mit konstanten Indices vorgenommen werden darf. Die praktischen Erfahrungen haben jedoch noch keinen darüber hinaus gehenden Bedarf erkennen lassen.

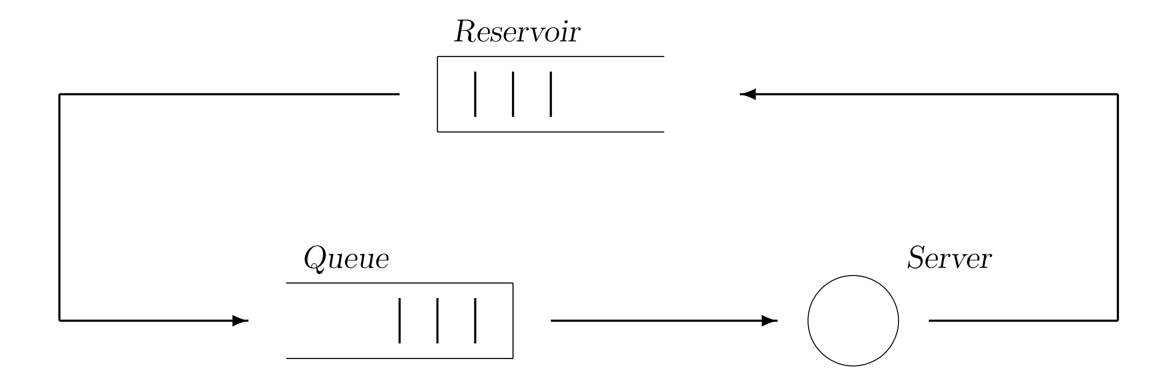
### 3.5.6 Probleme der Modularisierung

Eine der wichtigsten Anforderungen an eine deklarative Modellbeschreibungssprache war die Modularisierbarkeit von Modellen. Die klassische Systemtheorie lieferte hierfür einen besonders eleganten Ansatz. Wir wollen nun prüfen, inwieweit die bislang eingeführten sprachlichen Mittel für eine komponentenweise Zerlegung von Warteschlangenmodellen ausreichend sind.

Zu diesem Zweck modellieren wir das einfachste aller Warteschlangenmodelle, ein M/D/1-System.

Das Modell bestehe aus drei Aufenthaltsorten: einer Warteschlange `Queue`, einer Bedieneinheit `Server` und einem Vorrat von Elementen `Reservoir`.

Auf ein dynamisches Erzeugen und Vernichten von Elementen wird vorläufig bewußt verzichtet.




Anfangs halten sich alle mobilen Elemente im `Reservoir` auf. Mit einer exponentiell verteilten Zwischenankunftszeit `TBetween` betreten mobile Elemente die Warteschlange `Queue`. Ist die Bedieneinheit `Server` frei, wechselt das erste Element der `Queue` dorthin. Es hält sich dann während der konstanten Bedienzeit `TService` in der Bedieneinheit auf. Danach verläßt es den `Server` und wechselt in das `Reservoir` über.

Die Elemente gehören der Klasse `Job` an, deren Datenstruktur vereinbart ist durch:

```
MOBILE COMPONENT  Job

DEFINITION
   VALUE SET  Queues: ( 'Reservoir', 'Queue', 'Server' )

DECLARATIONS
   STATE VARIABLES    Ort (Queues)  := 'Reservoir' ,
                      TEntry (REAL) :=  0
END OF  Job
```

Es existiere ein Vorrat von N = 100 Elementen:

```
DIMENSION CONSTANT  N := 100

ARRAY {1..N}  Job
```

Die Dynamik des Modells läßt sich nun folgendermaßen beschreiben:

```
   FOREACH  i IN  1..N
   LET
#      Bewegung von Queue nach Server
#      ------------------------------
       IF  i  IN  {j IN 1..N  |  Job[j].Ort = 'Queue'
                              || INC  Job[j].TEntry}: {1}
       LET
           IF  EMPTY {j IN 1..N  |  Job[j].Ort = 'Server'}
           LET
               Job[i].Ort^    := 'Server';
               Job[i].TEntry^ :=  T;
           END
       END
```

```
#       Bewegung von Server nach Reservoir
#       ----------------------------------
        ELSIF  i  IN  {j IN 1..N  |  Job[j].Ort = 'Server'}
        LET
            IF  T >= Job[i].TEntry + TService
            LET
                Job[i].Ort^    := 'Reservoir';
                Job[i].TEntry^ :=  T;
            END
        END

#       Bewegung von Reservoir nach Queue
#       ---------------------------------
        ELSIF  i  IN  {j IN 1..N  |  Job[j].Ort = 'Reservoir'}: {1}
        LET
            IF  T >= TGenera
            LET
                Job[i].Ort^    := 'Queue';
                Job[i].TEntry^ :=  T;
                TGenera^       :=  T + TBetween;
            END
        END
    END
```

Die vorliegende Modellbeschreibung genügt unseren Anforderungen an die semantische Korrektheit. Insbesondere ist durch die Verwendung des IF-ELSIF-Konstrukts die Eindeutigkeit der Wertzuordnungen gewährleistet.

Eine sinnvolle Modularisierung wäre sicherlich, die Beschreibungen für die drei Transporte in separaten Komponenten unterzubringen. Zu diesem Zweck muß allerdings die obige, geschachtelte Sprachkonstruktion zerlegt werden.

Damit erlangt die Modellbeschreibung folgendes Aussehen:

```
#   Bewegung von Queue nach Server
#   ------------------------------
    FOREACH  i IN  1..N
    LET
        IF  i  IN  {j IN 1..N  |  Job[j].Ort = 'Queue'
                               || INC  Job[j].TEntry}: {1}
        LET
            IF  EMPTY {j IN 1..N  |  Job[j].Ort = 'Server'}
            LET
                Job[i].Ort^    := 'Server';
                Job[i].TEntry^ :=  T;
            END
        END
    END
```

```
#  Bewegung von Server nach Reservoir
#  ---------------------------------
   FOREACH  i IN  1..N
   LET
       IF  i  IN  {j IN 1..N  |  Job[j].Ort = 'Server'}
       LET
           IF  T >= Job[i].TEntry + TService
           LET
               Job[i].Ort^    := 'Reservoir';
               Job[i].TEntry^ :=  T;
           END
       END
   END

#  Bewegung von Reservoir nach Queue
#  --------------------------------
   FOREACH  i IN  1..N
   LET
       IF  i  IN  {j IN 1..N  |  Job[j].Ort = 'Reservoir'}: {1}
       LET
           IF  T >= TGenera
           LET
               Job[i].Ort^    := 'Queue';
               Job[i].TEntry^ :=  T;
               TGenera^       :=  T + TBetween;
           END
       END
   END
```

Hiermit wäre nun die Voraussetzung geschaffen, um die Beschreibung der drei Bewegungen auf drei Komponenten zu verteilen. Leider hat diese Umformung den Preis, daß keine Eindeutigkeit mehr gewährleistet ist. Der menschliche Betrachter ist zwar schon in der Lage zu erkennen, daß die drei Indexmengen in den IF-Zweigen disjunkt sind, weil das Attribut `Ort` drei verschiedene Ausprägungen annimmt, ein Compiler jedoch kann diese Aufgabe im allgemeinen nicht übernehmen.

Um überhaupt eine Modularisierung durchführen zu können, verzichten wir vorläufig auf die Überprüfung der Eindeutigkeit zur Compile-Zeit und beschränken uns auf eine Fehlererkennung zur Laufzeit in einer konkreten, fehlerhaften Situation.

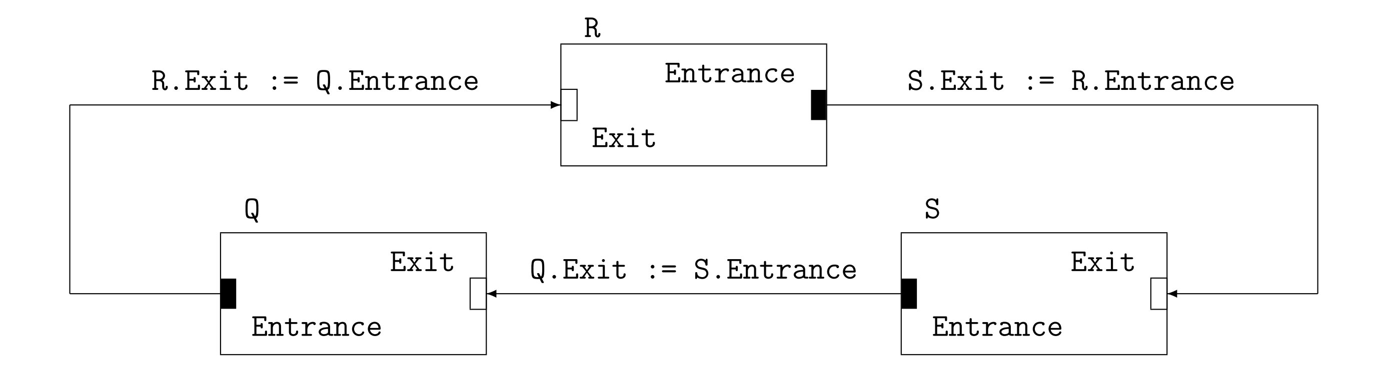
Der erste Versuch einer modularen Modellzerlegung nimmt folgende Gestalt an:

Da einer Komponente nicht bekannt ist, wohin es ein Element schicken soll, wenn der Ort in einer fremden Komponente liegt, sind Sensorvariablen einzuführen, um diese Informationen zu beschaffen.

Für jede Komponente wird eine Sensorvariable `Exit` und eine Zustandsvariable `Entrance` vereinbart.

Da nicht vorgesehen ist, daß Sensorvariablen mit Werten (z.B. `Server`) verbunden sind, müssen auch noch Zustandsvariable eingeführt werden, deren Aufgabe nur darin liegt, das Ziel, an welches das mobile Element zu verschicken ist, mitzuteilen.

```
EFFECT CONNECTIONS
Q.Entrance  -->  R.Exit;
S.Entrance  -->  Q.Exit;
R.Entrance  -->  S.Exit;
```




Das Bild zeigt eine Modellzerlegung in die drei Komponenten Q, S und R mit den erforderlichen Verbindungen. Die Modellbeschreibung für jede dieser drei Komponenten erlangt damit folgendes Aussehen:

```
#  ------------------------------------------------------------
#  Komponente 'Q'
#  ------------------------------------------------------------
   STATE VARIABLE   Entrance (Queues) := 'Queue'
   SENSOR VARIABLE  Exit (Queues)

#  Bewegung von Queue nach Exit (Server)
#  -------------------------------------
   FOREACH  i IN  1..N
   LET
       IF  i  IN  {j IN 1..N  |  Job[j].Ort = 'Queue'
                              || INC  Job[j].TEntry}: {1}
       LET
           IF  EMPTY {j IN 1..N  |  Job[j].Ort = Exit}
           LET
               Job[i].Ort^    :=  Exit;
               Job[i].TEntry^ :=  T;
           END
       END
   END
```

```
#  ------------------------------------------------------------
#  Komponente 'S'
#  ------------------------------------------------------------
   STATE VARIABLE   Entrance (Queues) := 'Server'
   SENSOR VARIABLE  Exit (Queues)

#  Bewegung von Server nach Exit (Reservoir)
#  ----------------------------------------
   FOREACH  i IN  1..N
   LET
       IF  i  IN  {j IN 1..N  |  Job[j].Ort = 'Server'}
       LET
           IF  T >= Job[i].TEntry + TService
           LET
               Job[i].Ort^    :=  Exit;
               Job[i].TEntry^ :=  T;
           END
       END
   END

#  ------------------------------------------------------------
#  Komponente 'R'
#  ------------------------------------------------------------
   STATE VARIABLE   Entrance (Queues) := 'Reservoir'
   SENSOR VARIABLE  Exit (Queues)

#  Bewegung von Reservoir nach Exit (Queue)
#  ---------------------------------------
   FOREACH  i IN  1..N
   LET
       IF  i  IN  {j IN 1..N  |  Job[j].Ort = 'Reservoir'}: {1}
       LET
           IF  T >= TGenera
           LET
               Job[i].Ort^    :=  Exit;
               Job[i].TEntry^ :=  T;
               TGenera^       :=  T + TBeteen;
           END
       END
   END
```

Betrachten wir das Beispiel genauer, dann stellen wir eine ganze Reihe von Unzulänglichkeiten fest.

1. Die Wertemenge der Variablen `Ort`, d.h. alle möglichen Aufenthaltsorte eines Elements, muß bereits bei der Deklaration der mobilen Komponente `Job` vollständig bekannt sein.

2. Jede Komponente hat über den Index prinzipiell Zugang zu allen mobilen Elementen einer Klasse. Eine komponentenweise Zerlegung ergibt jedoch nur dann einen Sinn, wenn eine Komponente auf einen Teil ihrer Zugriffsmöglichkeiten verzichtet. Hierfür gibt es aber bislang keine Vorkehrungen.

3. Das Klassenkonzept kann nicht eingesetzt werden, da alle Instanzen einer Klasse dieselben Aufenthaltsorte verwenden würden.
4. Die Überprüfbarkeit der Eindeutigkeit durch den Compiler mußte aufgegeben werden.
5. Die Zustandsvariable `Entrance` mutet künstlich an, da sie nur der Weitergabe eines einzelnen Wertes dient.
6. Die Pfeile, welche die Komponenten miteinander verbinden, verlaufen entgegen der Transportrichtung und dienen damit nicht dem Verständnis des Modells auf höherer Ebene.

Wie wir feststellen müssen, handelt es sich hier nicht mehr um eine Modularisierung im ursprünglichen Sinne.

Durch eine weitere Ergänzung des Sprachumfangs wollen wir versuchen, diese Nachteile zu beseitigen. Am aussichtsreichsten erscheint die Einführung von Mengenvariablen, die bereits geordnete Warteräume, d.h. Warteschlangen repräsentieren. Die Inhalte dieser Mengenvariablen könnten dann auch anderen Komponenten zugänglich gemacht werden. Außerdem versprechen wir uns davon auch eine kompaktere Beschreibung des Modells und die Möglichkeit, die zu einer Warteschlange gehörigen Elemente gemeinsam beobachten und ausgeben zu können.

## 3.6 Einführung von Mengenvariablen als abhängige Variablen

Im zweiten Ansatz formulieren wir Warteschlangenmodelle, indem wir Mengen von mobilen Elementen bilden und diese Mengen abhängigen Variablen zuordnen.

### 3.6.1 Variablen für Indexmengen

Die bisherigen Operationen dienten lediglich der Bildung von Mengen. Wir wollen nun prüfen, ob es sich als vorteilhaft erweist, Mengen auch als abhängige Variablen zuzulassen. Bei der Deklaration wird anstelle der Wertemenge das Schlüsselwort INDEXSET angegeben.

Beispiel:
```
DEPENDENT VARIABLE  Queue1 (INDEXSET), Queue2 (INDEXSET)
```

Die Belegung erfolgt durch Zuordnung einer Indexmenge.

Variablen, denen eine Menge zugeordnet wird, bezeichen wir als Mengenvariablen und unterscheiden sie dadurch von den Eigenschaftsvariablen, denen ein einzelner Wert zugeordnet wird.

Beispiele:
```
Queue1 :=  {1..10};
Queue1 :=  {i IN 1..10 | X[i] > 0};
Queue1 :=  {i IN 1..10 | Job[i].Ort = 'Q1'};
```

Eine Mengenvariable kann auch für eine geordnete Menge stehen.

Beispiel: `Queue1 := {i IN 1..10 | Job[i].Ort = 'Q1' || INC Job[i].Laenge};`

Eine Mengenvariable muß nicht unbedingt für einen Aufenthaltsort stehen. Die Bedingung ist schließlich beliebig und auch Elemente mehrerer Aufenthaltsorte können durch eine Mengenvariablen zusammengefaßt sein.

Beispiel: `Queue12 := {i IN 1..10 | Job[i].Ort = 'Q1' OR Job[i].Ort = 'Q2'};`

Aber auch fragwürdige Mengenbildungen sind möglich. So ist es denkbar, daß eine Mengenvariable Indices von mobilen Elemente verschiedenen Typs beinhaltet.

Beispiel: `QType12 := {i IN 1..10 | Job1[i].Ort = 'Q1' OR Job2[i].Ort = 'Q2'};`

Eine solche Mengenbildung ist in der Regel jedoch nicht sinnvoll, da jeder Index nur einmal in die Indexmenge aufgenommen wird. Die Elemente, die zur Bildung der Menge beigetragen haben, lassen sich über die Mengenvariable nicht mehr ansprechen.

Bildet man nicht eine Indexmenge, sondern eine Menge mobiler Elemente, wird dieses Problem vermieden.

### 3.6.2 Variablen für Mengen aus Feldelementen

Wir wollen nun (geordnete) Mengen bilden, die sich aus Elementen eines Feldes zusammensetzen. Der Name des deklarierten Feldes dient gleichzeitig als Name der Grundmenge. Aus dieser Grundmenge kann durch Angabe von Eigenschaften eine Teilmenge gebildet werden.

```
ordered_property_set ::=  '{' identifier  IN  basis_set
                                [ '|' condition ]  [ '||' criteria_list ] '}'

basis_set ::=   index_range
              | identifier
```

Beispiel:

```
ARRAY {1..N} Job

{J IN Job | J.Ort = 'Q1' || J.TEntry}
```

Auch für Mengen aus Feldelementen lassen sich Mengenvariablen einführen. Die Deklaration enthält als Wertemenge den Namen des Feldes.

Beispiel: `DEPENDENT VARIABLE  Queue1 (SET OF Job), Queue2 (SET OF Job)`

Zuordnungen an diese Mengenvariablen erfolgen in gewohnter Weise.

Beispiel: `Queue1 := {J IN Job | J.Ort = 'Q1' || INC J.Laenge};`

Mengen aus Feldelementen können beschnitten werden, indem man einen Indexbereich in Verbindung mit dem Namen des Feldes angibt. Diese Art der Teilmengenbildung kann sowohl auf geordnete Eigenschaftsmengen als auch auf geordnete Mengenvariablen angewendet werden.

```
cutted_set ::=   set  [ ':' [ identifier ]  cardinal_set ]

set        ::=   ordered_property_set
               | indexed_identifier
```

Beispiel: `{J IN Job | Job.Ort = 'Q1' || INC Job.TEntry} : Job{1..5}`

Die ersten fünf Elemente aus linksstehender Menge werden ausgegrenzt.

Beispiel:

```
Queue1:Job{1..5}      # Beschneiden einer Mengenvariablen
Queue1:Job{1}         # Teilmenge aus einem einzelnen oder
                      # gar keinem Element
```

Der Zugriff auf ein Attribut eines mobilen Elements, erfolgt in ähnlicher Weise. Der Index des mobilen Elements steht jedoch in eckigen Klammern. Damit wird vorausgesetzt, daß das angesprochene Element auch tatsächlich in der Menge enthalten ist.

```
selected_variable ::=  indexed_identifier ':' identifier  index
                                          '.' indexed_identifier
```

Beispiel:

```
Queue1:Job[1].Laenge    # Attribut Laenge des ersten Elements
                          in der Warteschlange Queue1
```

### 3.6.3 Beschreibung von Transporten

Die Einführung von Mengenvariablen verschafft uns eine verkürzte und übersichtlichere Modellbeschreibung.

Beispiel:

```
ARRAY {1..N}  Job

Queue1 := {J IN Job | J.Ort = 'Q1' ||  DEC J.Laenge, INC J.TEntry};

FOREACH  J  IN  Job
LET
    IF  CARD Queue1 >= 5
    LET
        IF  J IN Queue1:Job{1}
        LET
            J.Ort^    :=  'Q2';
            J.TEntry^ :=  T;
        END
    END
END
```

Wie man sieht, wurde auch der Allquantor so erweitert, daß als gebundene Variable nicht nur Indices, sondern auch Feldelemente in Frage kommen.

Es fällt auf, daß der Aufenthaltsort eines mobilen Elements einerseits als Menge (`Queue1`), andererseits als Wert einer Aufzählungsmenge (`'Q1'`) geführt wird. Um dies zu vermeiden, ordnen wir der Variable `Ort` eines mobilen Elements in Zukunft nicht mehr den Wert einer Aufzählungsmenge, sondern den Namen einer Mengenvariablen zu.

### 3.6.4 Namen von Mengen

Bezeichner stehen in der Regel für den Inhalt (Wert oder Menge) einer Variablen. Wir wollen nun auch Bezeichner zulassen, die für den Namen einer Variablen stehen. Der Name einer Variablen wird mit Hilfe des Operators NAME angesprochen.

Beispiel:

```
Ort (NAME(SET OF Job))          # Deklaration

J.Ort^ :=  NAME(Queue2);        # Wertzuordnung
```

Bei der Deklaration der Variablen Ort wird dazu als Wertemenge

```
NAME(SET OF identifier)
```

angegeben.

Als inverse Operation ließe sich der Operator VAL einführen, der den Namen einer Variablen in den Wert umformt und für den gilt:

```
VAL(NAME(identifier) = identifier
```

Diese Operation wird jedoch im weiteren keine Verwendung finden.

### 3.6.5 Modellierung eines M/D/1-Systems

Anhand der Beschreibung eines MD/1-Systems wollen wir sehen, wie sich die Verbesserungen auswirken.

```
#  ------------------------------------------------------------------
   Komponente  Job
#  ------------------------------------------------------------------
   MOBILE COMPONENT  Job

   DECLARATIONS
      STATE VARIABLES    Ort (NAME(SET OF Job)) :=  NAME(Reservoir) ,
                         TEntry (REAL)          :=  0
   END OF  Job
```

```
# ----------------------------------------------------------------
   Komponente  MD1
# ----------------------------------------------------------------
   DIMENSION CONSTANT  N := 100

   ARRAY {1..N}  Job

   Queue     :=  { J IN Job | J.Ort = NAME(Queue)      ||  INC J.TEntry };
   Server    :=  { J IN Job | J.Ort = NAME(Server)     ||  INC J.TEntry };
   Reservoir :=  { J IN Job | J.Ort = NAME(Reservoir)  ||  INC J.TEntry };


   FOREACH  J  IN  Job
   LET
#      Bewegung von Queue nach Server
#      ------------------------------
       IF  J  IN  Queue:Job{1}
       LET
           IF  EMPTY Server
           LET
               J.Ort^    :=  NAME(Server);
               J.TEntry^ :=  T;
           END
       END

#      Bewegung von Server nach Reservoir
#      ----------------------------------
       ELSIF  J  IN  Server:Job{1}
       LET
           IF  T >= J.TEntry + TService
           LET
               J.Ort^    :=  NAME(Reservoir);
               J.TEntry^ :=  T;
           END
       END

#      Bewegung von Reservoir nach Queue
#      ---------------------------------
       ELSIF  J  IN  Reservoir:Job{1}
       LET
           IF  T >= TGenera
           LET
               J.Ort^    :=  NAME(Queue);
               J.TEntry^ :=  T;
               TGenera^  :=  T + TBetween;
           END
       END
   END
```

Das Beispiel deckt allerdings noch einige Schwierigkeiten auf:

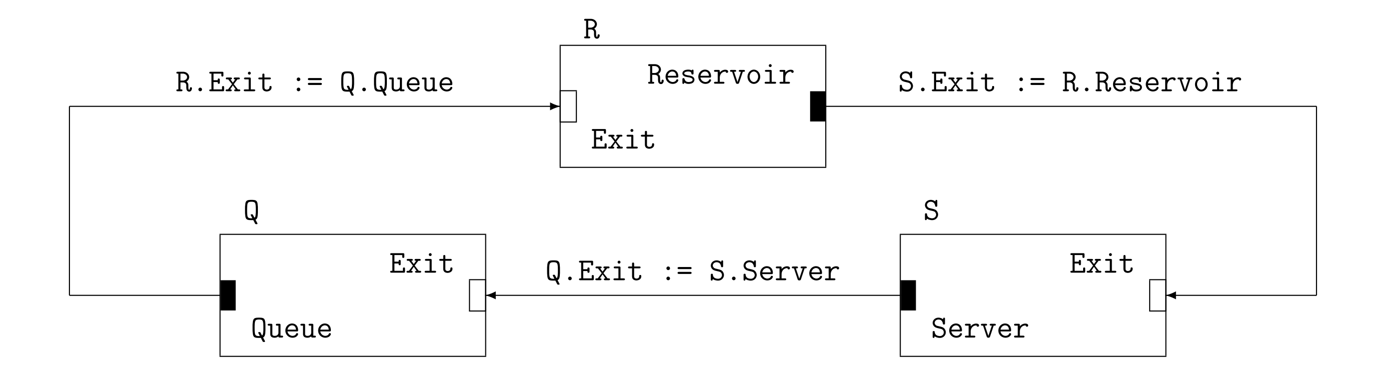
So stellt sich das Problem, wie eine Vorbesetzung des Ortes vorgenommen werden kann, da idealerweise bei der Deklaration der mobilen Elemente noch gar keine Aufenthaltsorte bekannt sind. Diese sollten erst innerhalb der (statischen) Komponenten vereinbart werden.

Ein anderes Problem ist, daß die Zustandsvariable 'Ort' nur dann aussagekräftig ist, wenn eine mobile Komponente stets genau einer der drei Mengen angehört, d.h. nicht gleichzeitig in einer anderen Menge als der zugewiesenen enthalten ist. Die Mengen, die einen Aufenthaltsort repräsentiern, müssen dazu disjunkt sein. Die korrekte Festlegung der Mengen ist jedoch vollständig dem Anwender überlassen.

### 3.6.6 Modularisierung mit Mengenvariablen

Durch die Einführung von Mengenvariablen ist es möglich geworden, Informationen über Mengen an andere Komponenten weiterzugeben.

```
EFFECT CONNECTIONS
Q.Queue      -->  R.Exit;
S.Server     -->  Q.Exit;
R.Reservoir  -->  S.Exit;
```




Das Bild zeigt eine Modellzerlegung in die drei Komponenten `Q`, `S` und `R`. Die Modellbeschreibung für jede dieser drei Komponenten erlangt damit folgendes Aussehen:

```
#  ---------------------------------------------------------------
#  Mobile Komponente  Job
#  ---------------------------------------------------------------
   MOBILE COMPONENT  Job

   DECLARATIONS
      STATE VARIABLES    Ort (NAME(SET OF Job)) :=  NAME(Reservoir) ,
                         TEntry (REAL)          :=  0
   END OF  Job
```

```
   DIMENSION CONSTANT  N := 100

   ARRAY {1..N}  Job

#  ------------------------------------------------------------
#  Komponente  Q
#  ------------------------------------------------------------
   DEPENDENT VARIABLE   Queue (SET OF Job)
   SENSOR VARIABLE      Exit  (SET OF Job)

   Queue :=  { J IN Job | J.Ort = NAME(Queue)  ||  INC J.TEntry };

#  Bewegung von Queue nach Exit (Server)
#  -------------------------------------
   FOREACH  J  IN  Job
   LET
       IF  J  IN  Queue:Job{1}
       LET
           IF  EMPTY Exit
           LET
               J.Ort^    :=  NAME(Exit);
               J.TEntry^ :=  T;
           END
       END
   END

#  ------------------------------------------------------------
#  Komponente 'S'
#  ------------------------------------------------------------
   DEPENDENT VARIABLE   Server (SET OF Job)
   SENSOR VARIABLE      Exit   (SET OF Job)

   Server :=  { J IN Job | J.Ort = NAME(Server)  ||  INC J.TEntry };

#  Bewegung von Server nach Exit (Reservoir)
#  -----------------------------------------
   FOREACH  J  IN  Job
   LET
       IF  J  IN  Server:Job{1}
       LET
           IF  T >= J.TEntry + TService
           LET
               J.Ort^    :=  NAME(Exit);
               J.TEntry^ :=  T;
           END
       END
   END
```

```
# ---------------------------------------------------------------
# Komponente 'R'
# ---------------------------------------------------------------
  DEPENDENT VARIABLE   Reservoir (SET OF Job)
  SENSOR VARIABLE      Exit      (SET OF Job)

  Reservoir :=  { J IN Job | J.Ort = NAME(Reservoir)  ||  INC J.TEntry };

# Bewegung von Reservoir nach Exit (Queue)
# ----------------------------------------
  FOREACH  J  IN  Job
  LET
      IF  J  IN  Reservoir:Job{1}
      LET
          IF  T >= TGenera
          LET
              J.Ort^    :=  NAME(Exit);
              J.TEntry^ :=  T;
              TGenera^  :=  T + TBetween;
          END
      END
  END
```

Gegenüber dem ersten Ansatz konnten durch die Einführung der abhängigen Mengenvariablen zwei wesentliche Mängel beseitigt werden:

Vor allem ist der Gebrauch des Klassenkonzepts weiterhin gewährleistet. Bei einer Vervielfältigung der Komponenten vervielfältigen sich auch die Mengenvariablen bzw. Warteschlangen. Jede Mengenvariable erhält ihren individuellen Namen, der sich aus Komponenten- und Variablennamen zusammensetzt.

Beispiel:

```
SUBCOMPONENTS
   Q1  OF CLASS  Q,
   Q2  OF CLASS  Q,  ...
```

Die Mengenvariablen in den Komponenten `Q1` und `Q2` besitzen die Namen `Q1/Queue` und `Q2/Queue`.

Zum Verbindungsaufbau sind nun auch keine Dummy-Variablen mehr erforderlich, da die Mengenvariablen selbst an die anderen Komponenten weitergegeben werden können.

Nach wie vor besteht aber das Problem, in welcher Weise der Aufenthaltort der mobilen Komponenten initialisiert werden soll. Bei ihrer Deklaration sind die Namen der Mengenvariablen noch nicht bekannt.

Auch die übrigen Unzulänglichkeiten konnten noch nicht beseitigt werden.

## 3.7 Einführung von Mengenvariablen als Zustandsvariablen

Im dritten Ansatz sehen wir vor, daß jede mobile Komponente standardmäßig stets einen bestimmten Aufenthaltsort besitzt und damit immer einer Menge zugeordnet ist. Der Auf-

enthaltsort wird dadurch zu einer besonderen Menge, die als Zustandsvariable angesehen werden kann.

### 3.7.1 Locations und mobile Komponenten

Um einen Bezug zu den bisherigen Ausführungen herzustellen, überlegen wir uns zunächst, wie abhängige Mengenvariablen zu modifizieren wären, wenn wir den Aufenthaltsort als standardmäßiges Attribut einer mobilen Komponente einführen.

- Jede mobile Komponente wird standardmäßig mit einer Zustandsvariablen `LOC` mit Wertemenge `NAME(SET OF mobile_component)` versehen, die dem Anwender verborgen bleibt und auch nicht durch Wertzuordnung vorbelegt werden kann. Dieser Zustandsvariablen wird der momentane Aufenthaltsort der mobilen Komponente zugewiesen.

  Beispiel: `STATE VARIABLE   LOC (NAME(SET OF Job))`

- Wir führen abhängige Mengenvariablen mit Wertemenge `SET OF mobile_element` ein und ermöglichen bei deren Deklaration die Angabe von Kriterien, nach denen die mobilen Elemente anzuordnen sind. Diese Mengenvariablen stehen für mögliche Aufenthaltsorte einer mobilen Komponente, werden daher als Locations bezeichnet und können vom Benutzer deklariert werden.

  Beispiel: `DEPENDENT VARIABLE  Queue1 (SET OF Job  ||  DEC Priority, INC TEntry)`

  Die Definition einer Location erfolgt implizit aufgrund der bei der Deklaration gemachten Angaben.

  Beispiel: `Queue1 :=  {  J IN Job  |  J.LOC = NAME(Queue1)  }`

- Mobile Komponenten werden nicht mehr durch die Vereinbarung eines globalen Feldes angelegt. Stattdessen kann jede Location mit einer bestimmten Anzahl von mobilen Elementen vorbelegt werden.

  Beispiel: `Queue1 :=  5 Job;`

  Dadurch wird deren Standardattribut `LOC` entsprechend besetzt. Aus den Vorbelegungen aller Aufenthaltsorte ergibt sich die gesamte Menge an mobilen Elementen im Modell.

- Zugriffe auf mobile Komponenten sind nun nicht mehr global möglich, sondern nur noch über ihren Aufenthaltsort.

  Beispiele:

  | | |
  |---|---|
  | `Job[5]` | ist nicht mehr möglich |
  | `Queue:Job[1]` | erster Job auf Queue |
  | `Queue:Job{1..5}` | Menge mit den ersten fünf Jobs auf Queue (soweit vorhanden) |

- Die Zuordnung des Aufenthaltsortes wird durch einen speziellen Operator vorgenommen. Diesen Operator `'->'` bezeichnen wir als Transportoperator, weil er den Wechsel des Aufenthaltsortes und damit einen Transport zum Ausdruck bringt.

  Beispiel:

  `Queue1:Job[1] -> Queue2;` entspricht `Queue1:Job[1].LOC^ := NAME(Queue2);`

  Die mobile Komponente `Job[1]` auf `Queue1` befindet sich im folgenden Zustand auf Location `Queue2`.

  Der Transportoperator verbirgt das Standardattribut `LOC` eines mobilen Elements. Er ermöglicht lediglich einen schreibenden Zugriff auf die Variable `LOC`. Da Zugriffe auf mobile Komponenten nur noch über deren Aufenthaltsort möglich sind, wird ein lesender Zugriff auf das Attribut `LOC` überflüssig.

  Beispiel: Liefert `LOCATION (.)` den Aufenthaltort einer mobilen Komponente,
  dann gilt: `LOCATION (Queue1:Job[1])` ergibt `Queue1`

Jedes mobile Element erhält durch diese Maßnahmen umkehrbar eindeutig eine bestimmte Position innerhalb einer bestimmten Warteschlange. Da für den Benutzer der Aufenthaltsort als Attribut einer mobilen Komponente nicht mehr sichtbar ist, enthalten die Mengenvariablen (Locations) die Information über den Modellzustand.

Daher stellen sich die getroffenen Maßnahmen für den Benutzer folgendermaßen dar:

- Wir führen Zustandsvariablen mit Wertemenge `SET OF mobile_element` ein und ermöglichen bei deren Deklaration die Angabe von Ordnungskriterien bzw. Einordnungskriterien. Diese Mengenvariablen stehen für mögliche Aufenthaltsorte einer mobilen Komponente, werden daher als Locations bezeichnet und können vom Benutzer deklariert werden.

  Beispiel:

  ```
  STATE VARIABLE
        Queue1 (SET OF Job  ||  DEC Priority, INC TEntry) :=  3 Job
  ```

  Locations können mit einer bestimmten Anzahl von mobilen Komponenten vorbelegt werden. Aus den Vorbelegungen aller Aufenthaltsorte ergibt sich die gesamte Menge an mobilen Elementen im Modell.

- Zugriffe auf mobile Komponenten sind über ihren Aufenthaltsort möglich. Der Aufenthaltsort setzt sich aus der Location und der Position auf der Location zusammen.

  Beispiele:

  | | |
  |---|---|
  | `Queue:Job[1]` | erster Job auf Queue |
  | `Queue:Job{1..5}` | Menge mit den ersten fünf Jobs auf Queue (soweit vorhanden) |

- Ein Wechsel des Aufenthaltsortes wird durch einen speziellen Transportoperator `'->'` vorgenommen.

  Beispiel:

  ```
  Queue1:Job[1] -> Queue2;
  ```

  Die mobile Komponente `Job[1]` auf `Queue1` befindet sich im folgenden Zustand auf Location `Queue2`. Ihre Einordnung und damit die Bestimmung der Position erfolgt entsprechend des Einordnungskriteriums.

Da mobile Komponenten nur noch über ihren Aufenthaltsort angesprochen werden können, sollte der Zugriff möglichst einfach gehalten sein. Deshalb ist es sinnvoll, den Aufenthaltsort von vornherein mit einer Ordnung zu versehen, d.h. als Warteschlange und nicht als Warteraum auszubilden. Beim Zugriff auf einzelne Elemente muß dann nicht erst eine Ordnung hergestellt werden.

Zu diesem Zweck wird der Location bereits bei ihrer Deklartion ein Ordnungskriterium bzw. ein Einordnungskriterium mitgegeben. Ein Ordnungskriterium ordnet die Mengenelemente gemäß ihrer aktuellen Attribute. Ein Einordnungskriterium legt die Position bei der Ankunft eines Teiles fest und verändert diese nur, wenn das Element nachrücken kann. Ändern sich Attribute von mobilen Komponenten, so ändern sich dadurch nicht ihre Positionen innerhalb der Warteschlange.

Es darf nicht übersehen werden, daß die Angabe des Ordnungs- bzw. Einordnungskriteriums bei der Deklaration einer Location beträchtliche Einschränkungen mit sich bringt. Bislang konnten die Ausdrücke beliebig sein, um die Ordnung einer Menge festzulegen und insbesondere auch Funktionen beinhalten. Ordnungs- bzw. Einordnungskriterien von Locations hingegen dürfen lediglich Variablen der mobilen Komponente enthalten. Es ist daher nicht moglich, beliebige Strategien für die Selektion und die Entnahme mobiler Komponenten aus Warteschlangen anzugeben.

Bei der Wahl, ob Locations mit einem Ordnungs- oder einem Einordnungskriterium ausgestattet werden sollen, entscheiden wir uns für das leichter realisierbare Einordnungskriterium. Zustandsvariablen verharren im Zustand der Ruhe, wenn nicht ausdrückliche Operationen eine Zustandsänderung herbeiführen. Das allgemeinere Ordnungskriterium würde solche Zustandsübergänge implizieren. Da weder mit der einen noch mit der anderen Lösung alle Anforderungen erfüllt werden können, muß die Sprache ohnehin mit weiteren Konstrukten ausgestattet werden.

Diese Anforderungen lassen sich entweder durch die Einführung geordneter Eigenschaftsmengen (Kapitel 3.5) oder durch Funktionen erfüllen, für die Locations als Parameter zulässig sind und die einen Index zurückliefern. Dieses Thema wird an späterer Stelle (Kapitel 6) nochmals kurz angeschnitten.

### 3.7.2 Modellierung eines M/D/1-Systems

Das folgende Beispiel (wiederum das M/D/1-System) zeigt, daß die Einführung des Transportoperators dafür sorgt, daß der Operator NAME für den Benutzer verborgen bleibt, und sich eine besonders gut lesbare und kompakte Modellbeschreibung ergibt.

```
MOBILE COMPONENT  Job

DECLARATION
   STATE VARIABLE    TEntry (REAL)  :=  0
END OF  Job


BASIC COMPONENT  MD1

DECLARATIONS
   CONSTANT         TService  (REAL) := 0
   STATE VARIABLES  TGenera   (REAL) := 0 ,
                    Queue     (SET OF Job || INC TEntry) :=    0 Job ,
                    Server    (SET OF Job)               :=    0 Job ,
                    Reservoir (SET OF Job)               :=  100 Job
   RANDOM VARIABLE  TBetween  (REAL) : EXPO (Mean := 10)

DYNAMIC BEHAVIOUR

#  Bewegung von Queue nach Server
#  ------------------------------
   FOREACH  J  IN  Queue:Job{1}
   LET
       IF  EMPTY Server
       LET
           J -> Server;
           J.TEntry^ :=  T;
       END
   END

#  Bewegung von Server nach Reservoir
#  ----------------------------------
   FOREACH  J  IN  Server:Job{1}
   LET
       IF  T >= J.TEntry + TService
       LET
           J -> Reservoir;
           J.TEntry^ :=  T;
       END
   END
```

```
#  Bewegung von Reservoir nach Queue
#  --------------------------------
   FOREACH  J  IN  Reservoir:Job{1}
   LET
       IF  T >= TGenera
       LET
           J -> Queue;
           J.TEntry^ :=  T;
           TGenera^  :=  T + TBetween;
       END
   END
```

Da sichergestellt ist, daß eine mobile Komponente nur in einer einzigen Location enthalten ist, darf die Beschreibung des Modellverhaltens auf mehrere FOREACH-Konstrukte verteilt werden, ohne daß dadurch die Eindeutigkeit verletzt werden könnte.

### 3.7.3 Modularisierung mit Locations

Eine modulare Zerlegung dieses Modells ist nun fast ohne Abstriche an die gestellten Anforderungen möglich.

```
#  -----------------------------------------------------------
#  Mobile Komponente  Job
#  -----------------------------------------------------------
   MOBILE COMPONENT  Job

   DECLARATIONS
      STATE VARIABLE    TEntry (REAL)  :=  0
   END OF  Job


#  -----------------------------------------------------------
#  Komponente  Q
#  -----------------------------------------------------------
   STATE  VARIABLE      Queue (SET OF Job || INC TEntry) :=  0 Job
   SENSOR VARIABLE      Exit  (SET OF Job)

#  Bewegung von Queue nach Exit (Server)
#  ------------------------------------
   FOREACH  J  IN  Queue:Job{1}
   LET
       IF  EMPTY Exit
       LET
           J -> Exit;
           J.TEntry^ :=  T;
       END
   END
```

```
# ---------------------------------------------------------------
# Komponente 'S'
# ---------------------------------------------------------------
  CONSTANT            TService (REAL) := 8

  STATE VARIABLE      Server (SET OF Job) :=  0 Job
  SENSOR VARIABLE     Exit   (SET OF Job)

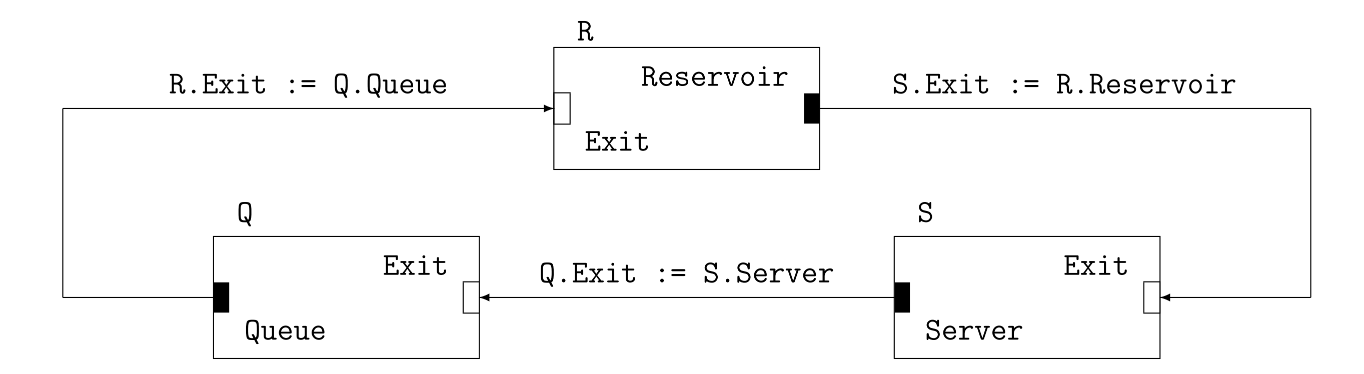
# Bewegung von Server nach Exit (Reservoir)
# ----------------------------------------
  FOREACH  J  IN  Server:Job{1}
  LET
      IF  T >= J.TEntry + TService
      LET
           J -> Exit;
           J.TEntry^ :=  T;
      END
  END

# ---------------------------------------------------------------
# Komponente 'R'
# ---------------------------------------------------------------
  STATE VARIABLE      TGenera   (REAL) := 0
                      Reservoir (SET OF Job) :=  100 Job
  SENSOR VARIABLE     Exit      (SET OF Job)
  RANDOM VARIABLE     TBetween  (REAL) : EXPO (Mean := 10)


# Bewegung von Reservoir nach Exit (Queue)
# ---------------------------------------
  FOREACH  J  IN  Reservoir:Job{1}
  LET
      IF  T >= TGenera
      LET
           J -> Exit;
           J.TEntry^ :=  T;
           TGenera^  :=  T + TBetween;
      END
  END

# ---------------------------------------------------------------
# Hoehere Komponente 'H'
# ---------------------------------------------------------------
  SUBCOMPONENTS  Q, S, R

  EFFECT CONNECTIONS
     Q.Queue      -->  R.Exit;
     S.Server     -->  Q.Exit;
     R.Reservoir  -->  S.Exit;
```




Unschön bleibt lediglich, daß die Pfeile, welche die Komponenten verbinden, nach wie vor entgegen der Transportrichtung weisen. Aber auch dieses Problem wird im nächsten Kapitel behoben.

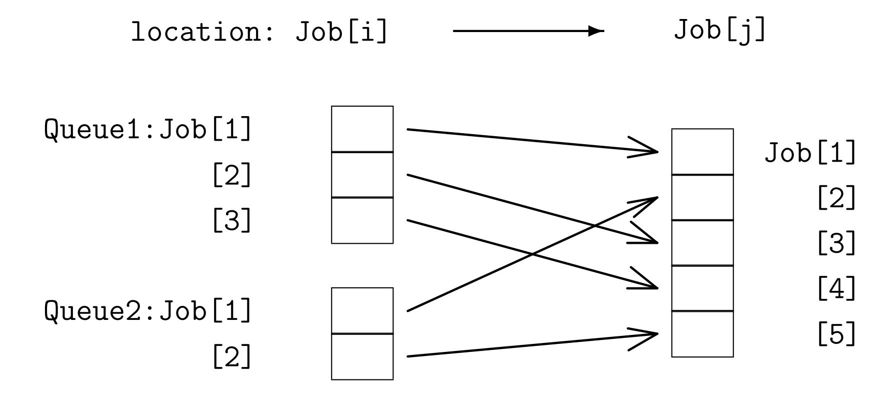
### 3.7.4 Methodologische Gesichtspunkte

Anfänglich wurde der Aufenthaltsort einer mobilen Komponente durch eine ihrer Zustandsvariablen bestimmt. Eine Warteschlange wurde als Menge mobiler Komponenten aufgefaßt, bei denen diese Zustandsvariable den gleichen Wert besitzt. Mengenvariablen, die durch Zuordnung einer derartigen (geordneten) Eigenschaftsmenge definiert wurden und Warteschlangen repräsentierten, waren daher abhängige Variablen.

Mit der Einführung der Location wurde der mobilen Komponente die Zustandsvariable entzogen, die den Aufenthaltsort bestimmt. Diese Information über den Modellzustand wird nun durch die Inhalte der Locations festgelegt.

Im Sinne der Informatik verkörpern Locations Variablen, die Listen von Modellobjekten (mobilen Komponenten) verkörpern und nicht wie bisher für singuläre Werte stehen. Die traditionelle systemtheoretische Beschreibungsform, die den Modellzustand allein durch Attribute beschreibt, wird damit verlassen und durch einen neuen Variablentyp, die geordnete Menge (Location), ergänzt.

Die Richtung der Zuordnung wurde umgekehrt. Bisher war jeder mobilen Komponente eindeutig ein Aufenthaltsort zugeordnet. Jetzt wird über eine Location und eine Position eindeutig auf eine mobile Komponente zugegriffen. Zugunsten einer modularen Zerlegbarkeit von Modellen ist ein direkter Zugriff auf mobile Komponenten nicht mehr möglich.




Eine Location kann als eine Zustandsvariable aufgefaßt werden, die den Aufenthaltsort von mobilen Komponenten bestimmt, indem sie eine Teilmenge aller mobilen Komponenten geordnet zusammenfaßt.

Man kann die nun entstandene Situation auch als Wechsel des Bezugssystems interpretieren. Während bisher das Modellgeschehen aus der Sichtweise der mobilen Elemente beschrieben wurde, wird es nun aus der Sicht eines übergeordneten, systemweiten Betrachters dargestellt.

Eine Transportoperation, d.h. ein Wechsel des Aufenthaltsortes, entspricht einer Zusammenfassung zweier Operationen auf Locations: einem Wegnehmen und einem Hinzufügen von mobilen Komponenten.

Es gilt die folgende Äquivalenz für einen Transport:

$$\texttt{Q1:Job[1] -> Q2} \quad \longleftrightarrow \quad \left\{ \begin{array}{llll} \texttt{Q2\^{}} & \texttt{:=} & \texttt{Q2} \cup & \texttt{Q1:Job[1];} \\ \texttt{Q1\^{}} & \texttt{:=} & \texttt{Q1} \setminus & \texttt{Q2:Job[1];} \end{array} \right.$$

Diese Äquivalenz läßt nochmals deutlich erkennen, daß es sich bei Locations in der Tat um Zustandsvariablen handelt.

Findet der Transport über Komponentengrenzen hinweg statt, so bedeutet dies, daß eine mobile Komponente auf der Location einer fremder Komponente plaziert wird. Dieser direkte Eingriff in den Zustand einer fremden Komponente bedeutet einen Verlust an Autonomie, mit dem wir uns im nächsten Kapitel auseinandersetzen werden.

Transporte zwischen Locations ermöglichen eine Beschreibung von Warteschlangenmodellen, die auf abhängige Mengenvariablen ganz verzichtet. Auch die Bildung von Eigenschaftsmengen ist nur noch in Ausnahmefällen nötig, da es meist genügt, Teilmengen aus Locations auszugrenzen. Der Sprachumfang kann damit verhältnismäßig klein gehalten werden.

Locations garantieren auch eine hohe Effizienz bei der Simulation der Modelle. Das Entnehmen bzw. Einreihen einer mobilen Komponente sind Operationen, die nur bei einem Transport ausgeführt werden müssen. Dazu brauchen nur einzelne Elemente aus- und eingekettet werden. Abhängige Mengenvariablen oder Eigenschaftsmengen hingegen müßten jedesmal vollständig neu bestimmt bzw. geordnet werden, wenn sich für ein mobiles Element der Grundmenge der Wert eines Attributs ändert. Die im letzten Kapitel diskutierten Maßnahmen zur Effizienzverbesserung können in vollem Umfang genutzt werden.

## 3.8 Zusammenfassung

Zur Behandlung von Warteschlangensystemen wurde die bisherige Sprachsyntax durch eine Reihe von neuen Datentypen und Konstrukten ergänzt.

Elemente bzw. mobile Komponenten dienen dazu, die Aufträge oder Teile zu modellieren, die sich in Warteschlangen und Bedienstationen aufhalten.

Da in einem Modell eine ganze Vielzahl solcher Elemente enthalten sind und jedes dieser Elemente das gleiche dynamische Verhalten zeigt, führten wir Felder ein, um viele Ausprägungen eines Elements zu erzeugen.

Definitionsgleichungen, welche die Belegung der Attribute bestimmen, sollen nunmehr auch für eine größere Anzahl von Feldindices gelten können. Wir betteten dazu Definitionsgleichungen in das FOREACH-Konstrukt ein, welches eine Art Allquantor darstellt.

Es stellte sich heraus, daß die Gewährleistung der Eindeutigkeit und Vollständigkeit im Zusammenhang mit Feldern neue Betrachtungen erfordert. Werden als Index Modellvariablen

oder Ausdrücke verwendet, dann ist die semantische Korrektheit eines Modells gefährdet. Durch geeignete Einschränkungen lassen sich diese Probleme lösen.

Mit diesen naheliegenden sprachlichen Erweiterungen wurde nun versucht, Teile von Warteschlangenmodellen zu formulieren. Es stellte sich heraus, daß für die Entnahme von Elementen aus Warteschlangen die Bildung von geordneten Mengen erforderlich ist, wenn man den dazu erforderlichen Aufwand in Grenzen halten will.

Während es nun zufriedenstellend möglich war, Warteschlangenmodelle zu formulieren, erwies sich die Modularisierung solcher Modelle als unzulänglich. Durch die Einführung von Mengenvariablen, deren Namen auch an andere Komponenten weitergegeben werden können, wurde ein Teil dieser Probleme überwunden.

Unbefriedigend war aber nach wie vor die Initialisierung des Aufenthaltsortes von mobilen Elementen sowie der ungeregelte Zugang zu mobilen Komponenten, die gar nicht in der betrachteten Modellkomponente enthalten sind. Diesen Problemen begegneten wir durch die Einführung von Locations, d.h. Mengen von mobilen Komponenten, die als Zustandsvariablen zu verstehen sind. Hierdurch wurden auch die letzten Schwierigkeiten beseitigt.

Ganz bewußt wurden die Erweiterungen der Sprache Zug um Zug durchgeführt, damit der Leser die Lösungsvorschläge gegeneinander abwägen kann. Am Ende des letzten Abschnitts zeigte sich, daß die Einführung von Locations einen großen Teil von bereits eingeführten Sprachkonstrukten wieder überflüssig macht. So erweist sich die Vereinbarung von abhängigen Mengenvariablen als überflüssig und auch die Bildung von Eigenschaftsmengen wird kaum noch benötigt. Lediglich die Ausgrenzung von Teilmengen ist für die Modellbildung unbedingt erforderlich. Diese Beschränkungen ermöglichen auch eine effiziente Simulation von Warteschlangenmodellen.

Als unbefriedigend wird allerdings empfunden, daß dieses Konzept zunächst nur auf die Bedürfnisse der Modellierung von Warteschlangen- und Transportmodellen zugeschnitten ist. Die Äquivalenz zwischen der Transportoperation und den beiden Teiloperationen Hinzufügen und Wegnehmen liefert uns jedoch eine Brücke zu kontinuierlichen und diskreten Modellen, die keine unterscheidbaren, mobilen Elemente beinhalten.

Anhand der Transportoperation wollen wir im folgenden Kapitel Gemeinsamkeiten und Unterschiede dieser bislang stets völlig getrennt behandelten Modellwelten herausarbeiten.

Ein weiterer wichtiger Punkt wird die weitgehende Wiederherstellung der Autonomie sein, die durch den Einsatz der Transportoperation in modular zerlegten Modellen verletzt wird. Damit verbunden ist auch die Tatsache, daß die Eindeutigkeit in zerlegten Modellen unter gewissen Bedingungen nicht gewährleistet ist.

# Kapitel 4

# Transporte zwischen Beständen

Im vorangegangenen Kapitel haben wir durch geeignete sprachliche Erweiterungen des systemtheoretischen Konzepts eine Beschreibung von Warteschlangenmodellen möglich gemacht. Der eleganteste Weg, der auch am besten die sematische Korrektheit eines Modells sicherstellen konnte, ergab sich durch die Einführung von Aufenthaltsorten (Locations), die man als Zustandsvariable interpretieren kann und die eine Menge von mobilen Komponenten repräsentieren.

Es wurde bereits erwähnt, daß sich die eingeführte Transportoperation ebenso als eine Kombination zweier Zustandsgleichungen auffassen läßt, einem Hinzufügen und einem Wegnehmen. Diese Art der Beschreibung ist typisch für die Formulierung von Transporten in kontinuierlichen Modellen oder diskreten Modellen ohne individuelle Elemente.

Dieses Kapitel stellt weitere Betrachtungen zur Transportoperation an. Es soll herausgefunden werden, ob eine einheitliche Beschreibung von Transporten möglich und sinnvoll ist.

Dabei dürfen dürfen die Anforderungen wie

- Eindeutigkeit der Beschreibung
- Erhaltung der mobilen Elemente
- nicht negative Bestände
- Autonomie der Komponenten bei modularer Zerlegung
- Verständlichkeit und Kompaktheit der Beschreibung

nicht aus den Augen verloren werden.

## 4.1 Motivation

Das folgende Beispiel - wiederum ein M/D/1-Modell - soll zeigen, wie man mit den klassischen Mitteln (siehe Kapitel 2) ein Warteschlangenmodell beschreiben kann, wenn man nicht zwischen einzelnen individuellen Elementen unterscheiden muß.

In diesem Fall läßt sich eine Warteschlange durch einen einfachen Zähler repräsentieren, der anzeigt, wieviele Elemente in der Warteschlange enthalten sind.

Es sei `NR` Anzahl Elemente im Reservoir
`NQ` Anzahl Elemente in der Queue
`NS` Anzahl Elemente im Server

Die Modelldynamik läßt sich hiermit folgendermaßen formulieren:

```
#  Bewegung von Queue nach Server
#  ------------------------------
   IF  NQ > 0  AND  NS = 0
   LET
       NQ^ :=  NQ - 1;
       NS^ :=  NS + 1;

       TEntry^ :=  T;
   END

#  Bewegung von Server nach Reservoir
#  ----------------------------------
   ELSIF  NS > 0  AND  T >= TEntry + TService
   LET
       NS^ :=  NS - 1;
       NR^ :=  NR + 1;
   END

#  Bewegung von Reservoir nach Queue
#  ---------------------------------
   ELSIF  T >= TGenera
   LET
       NR^ :=  NR - 1;
       NQ^ :=  NQ + 1;

       TGenera^ :=  T + TBetween
   END
```

Die Bewegungen werden hierbei durch Verändern der Bestände nachgebildet. Es obliegt dem Anwender, dafür zu sorgen, daß die Anzahl von Elementen, die einem Bestand entnommen wird, einem oder mehreren Beständen wieder zugeführt wird. Mit anderen Worten: Der Anwender ist dafür verantwortlich, daß die Anzahl der Elemente im Modell zu jedem Zeitpunkt erhalten bleibt.

Ebenso ist er dafür verantwortlich, daß aus einem Bestand keine Elemente entnommen werden, wenn keine darin enthalten sind. Bestände dürfen (falls nicht anders vereinbart) nicht negativ werden. Gegen beide Fehlerquellen kann die Sprache bei verteilter Beschreibung des Transports keinen Schutz gewährleisten.

Die Eindeutigkeit hingegen ist sichergestellt, denn durch die Verwendung des IF-ELSIF-Konstrukts können für die Zustandsvariablen `NR`, `NQ` und `NS` keine Mehrdeutigkeiten entstehen.

Dadurch wird die Problemetik des klassischen Erzeuger/Verbraucher-Modells vermieden, bei dem möglicherweise zum gleichen Zeitpunkt die Anzahl der der Elemente in der Warteschlange erhöht bzw. erniedrigt werden. Das IF-ELSIF-Konstrukt sorgt für den gegenseitigen Ausschluß des Zugriffs.

Für die beiden Ereignisse

```
IF  T >= TGenera                und           IF  NQ > 0  AND  NS = 0
LET                                           LET
    NQ^ :=  NQ - 1;                               NQ^ :=  NQ + 1;
END                                           END
```

```
   ------>  [ NQ ]  ------>
```

wäre dies nicht sichergestellt.

Es ist deshalb nicht ohne weiteres möglich, zwei Ereignisse, die beide auf den gleichen Bestand zugreifen, auf zwei verschiedene Komponenten zu verteilen.

Es gibt jedoch eine Methode, mit der sich eine Modellzerlegung unter Beibehaltung der Eindeutigkeit erreichen läßt: Hierzu ist die Einführung von abhängigen Variablen erforderlich, die anzeigen, wieviele Elemente gerade transportiert werden. Diese Methode ist sehr anschaulich und läßt sich bei Transporten zwischen Komponenten stets anwenden. Sie hat jedoch den Nachteil, daß sie das Modell wesentlich umfangreicher macht.

| Es sei | | |
|---|---|---|
| | `N_QS` | Anzahl Elemente, die sich von der Queue zum Server bewegen |
| | `N_SR` | Anzahl Elemente, die sich vom Server zum Reservoir bewegen |
| | `N_RQ` | Anzahl Elemente, die sich vom Reservoir zur Queue bewegen |

Ohne eine Modularisierung vorzunehmen, läßt sich die Modelldynamik hiermit folgendermaßen beschreiben:

```
#  Bewegung von Queue nach Server
#  ------------------------------
   IF  NQ > 0  AND  NS = 0
   LET
       N_QS :=  1;

       TEntry^ :=  T;
   END
   ELSE
   LET
       N_QS :=  0;
   END

#  Bewegung von Server nach Reservoir
#  ----------------------------------
   IF  NS > 0  AND  T >= TEntry + TService
   LET
       N_SR :=  1;
   END
   ELSE
   LET
       N_SR :=  0;
   END

#  Bewegung von Reservoir nach Queue
#  ---------------------------------
   IF  T >= TGenera
   LET
       N_RQ :=  1;

       TGenera^ :=  T + TBetween
   END
   ELSE
   LET
       N_RQ :=  0;
   END

#  Alle Bewegungen durchfuehren
#  ----------------------------
   NQ^ :=  N_Q  +  N_RQ - N_QS;
   NS^ :=  N_S  +  N_QS - N_SR;
   NR^ :=  N_R  +  N_SR - N_RQ;
```

Wie man sieht, sind keine ELSIF-Zweige mehr erforderlich, um die Eindeutigkeit herzustellen. Die Variablen lassen sich nun auf einzelne Komponenten verteilten und damit eine Modularisierung durchführen.

```
# -------------------------------------------------------------
# Komponente  Q
# -------------------------------------------------------------
  CONSTANT             TService (REAL) := 8
  STATE VARIABLE       NQ       (INT)  := 0
  DEPENDENT VARIABLE   N_QS     (INT)
  SENSOR VARIABLE      N_S      (INT) ,
                       N_RS     (INT)

# Transporte von Queue nach Server
# -------------------------------
  IF  NQ > 0  AND  NS = 0
  LET
      N_QS :=  1;
  END
  ELSE
  LET
      N_QS :=  0;
  END

# Aenderung in der Belegung der Queue
# ----------------------------------
  NQ^ :=  N_Q  +  N_RQ - N_QS;

# -------------------------------------------------------------
# Komponente 'S'
# -------------------------------------------------------------
  CONSTANT             TService (REAL) := 8
  STATE VARIABLE       TEntry   (REAL) := 0 ,
                       NS       (INT)  := 0
  DEPENDENT VARIABLE   N_SR     (INT)
  SENSOR VARIABLE      N_QS     (INT)

# Transporte von Server nach Reservoir
# -----------------------------------
  IF  NS > 0  AND  T >= TEntry + TService
  LET
      N_SR :=  1;
  END
  ELSE
  LET
      N_SR :=  0;
  END
```

```
#  Aenderung in der Belegung des Servers
#  -------------------------------------
   NS^ :=  N_S  +  N_QS - N_SR;

#  Eintrittszeitpunkt festhalten
#  -----------------------------
   IF  N_QS > 0
   LET
       TEntry^ := T;
   END

#  ------------------------------------------------------------
#  Komponente 'R'
#  ------------------------------------------------------------
   STATE VARIABLE       TGenera  (REAL) := 0 ,
                        NR       (INT)  := 0
   DEPENDENT VARIABLE   N_RQ     (INT)
   SENSOR VARIABLE      N_SR     (INT)
   RANDOM VARIABLE      TBetween (REAL) : EXPO (Mean := 10)

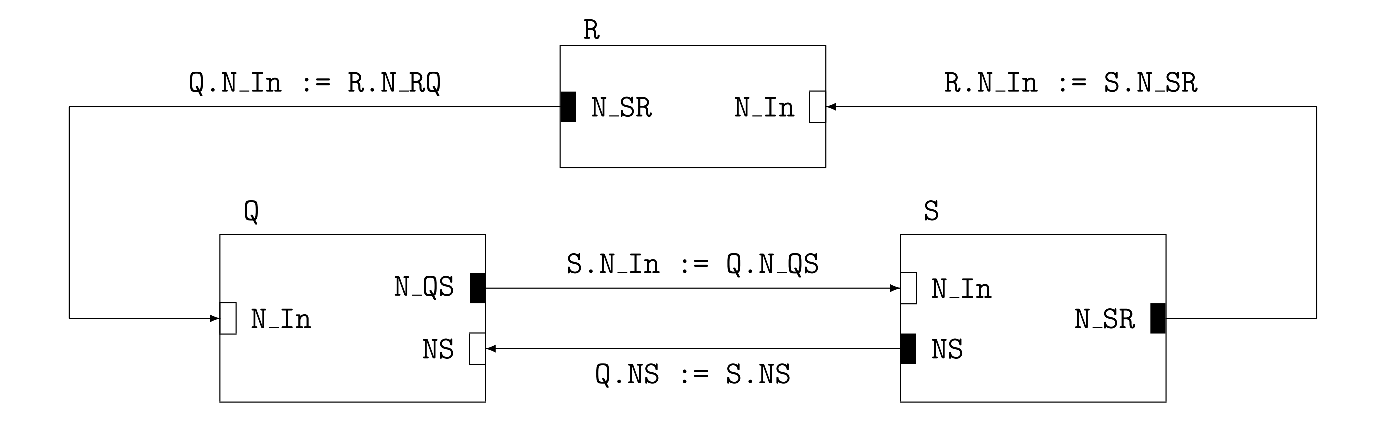
#  Transporte von Reservoir nach Queue
#  -----------------------------------
   IF  T >= TGenera
   LET
       N_RQ :=  1;

       TGenera^ :=  T + TBetween
   END
   ELSE
   LET
       N_RQ :=  0;
   END

#  Aenderung in der Belegung des Reservoirs
#  ----------------------------------------
   NR^ :=  N_R  +  N_SR - N_RQ;
```

```
#  -----------------------------------------------------------
#  Hoehere Komponente 'H'
#  -----------------------------------------------------------
   SUBCOMPONENTS  Q, S, R

   EFFECT CONNECTIONS
      Q.N_QS  -->  S.N_QS;        S.NS  -->  Q.NS;
      S.N_SR  -->  R.N_SR;
      R.N_RQ  -->  Q.N_RQ;
```




Verglichen mit dem entsprechenden Modell, das individuelle Elemente enthält und mit Transportoperationen beschrieben wurde, ist die Formulierung dieses Modells wesentlich umfangreicher und weniger leicht verständlich.

Es wurde auch bereits erwähnt, daß bei einer verteilten Beschreibung des Transports weder die Erhaltung der Elemente sichergestellt ist noch das Auftreten negativer Bestände ausgeschlossen werden kann.

Wegen dieser Nachteile wollen wir prüfen, ob die Einführung der Transportoperation auch für Modelle Vorteile bringt, die keine individuellen Elemente enthalten.

# 4.2 Bestände

## 4.2.1 Die Eigenheiten von Beständen

Die Einführung von Locations war mit der Forderung verbunden, daß jedem Element genau eine Menge zuzuordnen sei. Für die Mengen müssen hierzu zwei Voraussetzungen erfüllt sein:

- Die Mengen müssen disjunkt sein.
- Die Vereinigung aller Mengen muß die Gesamtheit aller Elemente umfassen.

Mengen, die diese Voraussetzungen erfüllen, bezeichnen wir als Bestände. Die Vereinigung aller Bestände bildet den Gesamtbestand des Modells.

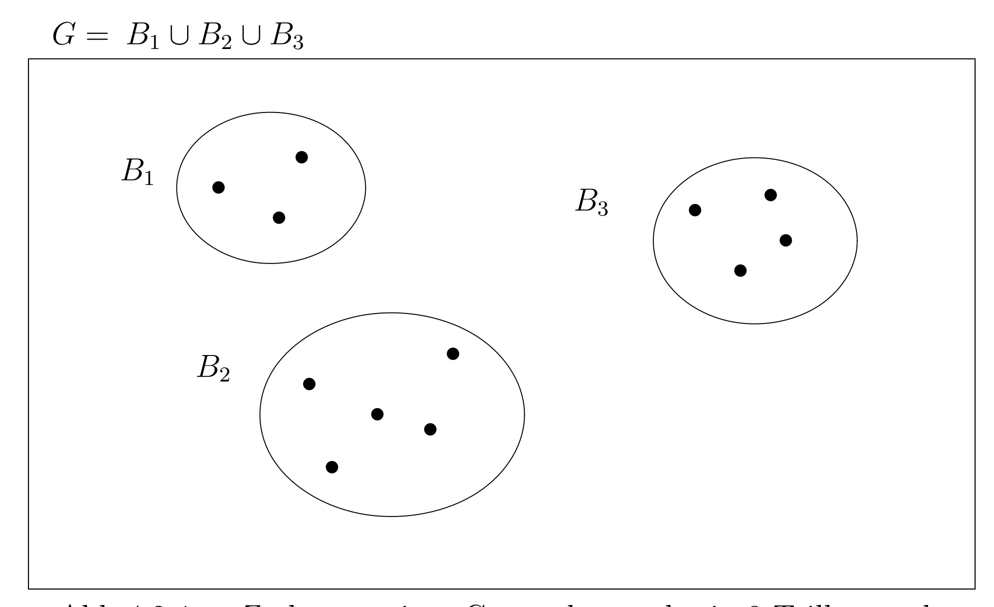



Abb 4.2-1: Zerlegung eines Gesamtbestandes in 3 Teilbestände

Ein Transport bewegt Elemente von einem Bestand zu einem anderen, ordnet diesen Elementen also einen anderen Bestand zu. Der Transport ist eine Operation, die den Gesamtbestand an Elementen im Modell nicht verändert. In einem abgeschlossenen Modell, d.h. in einem Modell, das weder Quellen noch Senken besitzt, in dem also Elemente weder erzeugt noch vernichtet werden, bleibt die gesamte Anzahl an Elementen in jedem Zeitpunkt (Zustand) konstant. In einem abgeschlossenen Modell gilt daher für die Elemente aller Bestände der Erhaltungssatz.

Die Zerlegung des Gesamtbestandes in disjunkte Teilmengen, d.h. in einzelne Bestände kann dabei nach Gesichtpunkten vorgenommen werden, die der Modellbildung dienlich sind. In der Regel faßt ein Bestand elementare Einheiten nach räumlichen oder logischen Gesichtspunkten zusammen.

Bestände kann man sich nicht nur aus einzelnen, individuellen Elementen zusammengesetzt vorstellen; vielmehr können sie auch durch eine positive ganze oder reelle Zahl beschrieben werden.

In seiner allgemeinen Form gehören einem Bestand eine Menge von Individuen an, die sich in ihren individuellen Eigenschaften unterscheiden.

Beispiele: Aufträge in Warteschlangen
Werkstücke in Maschinen, Puffern, Transporteinrichtungen

Ist für die Modellbildung eine Unterscheidung der Individuen eines Bestandes nicht erforderlich, benötigt man lediglich Variable, welche die Anzahl der Elemente eines Bestandes durch eine nichtnegative, ganze Zahl ausdrückt.

Beispiele: Schrauben in Magazinen
Kapital auf Bankkonten

Bei einer sehr großen Anzahl gleichartiger Individuen drückt man den Bestand häufig auch durch eine nichtnegative reelle Zahl aus, indem man in einer Maßeinheit eine festgelegte Zahl von Individuen zusammenfaßt.

Beispiele: Stoffmengen bei chemischen Reaktionen (in $mol$)
Volumina in der Hydraulik (in $m^3$)
Ladungsmengen in der Elektrotechnik (in $C$)

Eine Variable, die mit einer Maßeinheit verbunden ist, bezeichnet man in der Physik als Größe. Variablen, die einen Bestand repräsentieren bezeichen wir daher auch als Bestandsgrößen. Andere Variablen, die für ein Attribut stehen, nennen wir Eigenschaftsgrößen.

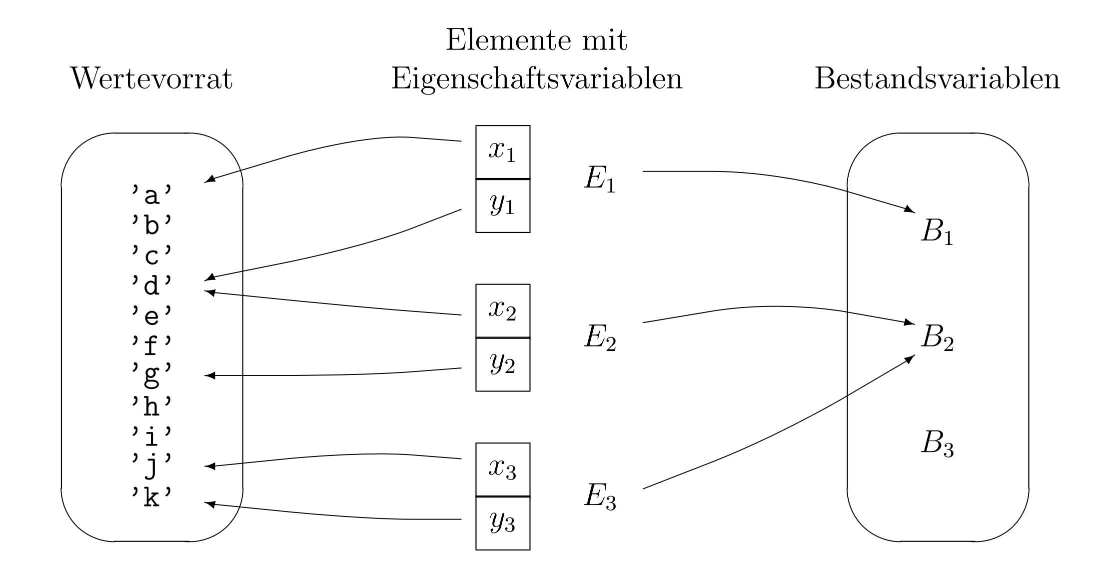
### 4.2.2 Bestandsgrößen und Eigenschaftsgrößen im Vergleich

Eine Eigenschaftsgröße, wie wir sie bisher kennengelernt haben, ist dadurch bestimmt, daß ihr aus einer vorgegebenen Wertemenge ein spezieller Wert eindeutig zugeordnet wird. Jeder Wert aus der Wertemenge hingegen, kann einer, mehreren oder auch gar keiner Eigenschaftsgröße, also einer Menge von Eigenschaftsgrößen zugeordnet sein.

Eine Bestandsgröße ist dadurch bestimmt, daß ihr aus einer Gesamtmenge eine Teilmenge von einzelnen Elementen zugeordnet wird. Umgekehrt läßt sich jedes einzelne Element eindeutig einer Bestandsgröße zuordnen.

Während eine Eigenschaftsgröße lediglich einen einzigen Wert annimmt, kann eine Bestandsgröße mehrere Elemente aufnehmen. Eine Eigenschaftsgröße wird daher von einer Wertvariablen, eine Bestandsgröße von einer Mengenvariablen repräsentiert. Die Definition einer Eigenschaftsgröße ist eine injektive, die Definition einer Bestandsgröße hingegen eine surjektive Abbildung.

Auch die Begriffe *Qualität* für Eigenschaftsgröße und *Quantität* für Bestandsgröße machen die Unterschiede deutlich.




Die Eigenschaftsvariablen $x_i$, $y_i$ sind zu Elementen $E_i$ zusammenfaßt.

| | | |
|---|---|---|
| Eigenschaftsvariable | $x_1 = \texttt{'a'}$ | Injektion |
| Bestandsvariable | $B_2 = \{E_2, E_3\}$ | Surjektion |

Für den Fall, daß eine Bestandsgröße lediglich die Anzahl von Elementen eines Bestandes repräsentiert, demnach eine ganze oder reelle Zahl ist, scheint kein Unterschied zwischen einer Bestandsgröße und einer Eigenschaftsgröße zu bestehen. Schließlich wird durch die Größe die Eigenschaft einer Menge, d.h. deren Kardinalität erfaßt.

Dies ist aber ein Trugschluß: Für Bestandsgrößen gilt der Erhaltungssatz, nicht jedoch für Eigenschaftsgrößen. Wird in einem abgeschlossenen Modell ein Bestand verändert, dann müssen die anderen Bestände in der Summe ebenfalls diese Änderung erfahren.

Beispiel: Entfernt man Masse von einer Stelle des Modells, dann muß dieselbe Masse an anderen Stellen im Modell wieder auftauchen.

Diesem Verhalten unterliegen Eigenschaftsgrößen nicht.

### 4.2.3 Bedeutung der Bestandsgrößen

Es läßt sich die interessante Beobachtung machen, daß sich die meisten empirischen Wissenschaften anhand der verwendeten Bestandsgrößen abgrenzen lassen. Bestandsgrößen spielen daher meist eine zentrale Rolle im Modell.

| | |
|---|---|
| Wirtschaftswissenschaften | Produktionsfaktoren: Kapital, Arbeitskräfte, Boden |
| Physik (allgemein) | Energie |
| Elektrotechnik | Ladungen |
| Hydraulik | Flüssigkeitsvolumina |
| Chemie | Stoffmengen |
| Soziologie | Menschen |

Allein aus der Modellvorstellung, daß ein Bestand eine Menge von Einheiten (Elementen) in einem abgegrenzten Raum darstellt, ergeben sich eine Reihe von abgeleiteten Eigenschaftsgrößen mit ihren Maßeinheiten:

Dichte
Konzentration — Menge von Einheiten pro Rauminhalt

$$[\text{ Dichte }] = \frac{[\text{ Menge }]}{[\text{ Volumen }]}$$

Strom — Menge von Einheiten durch eine Grenzfläche pro Zeit

$$[\text{ Strom }] = \frac{[\text{ Menge }]}{[\text{ Zeit }]}$$

Stromdichte — Menge von Einheiten durch eine Grenzfläche pro Zeit und Fläche

$$[\text{ Stromdichte }] = \frac{[\text{ Menge }]}{[\text{ Zeit }]\cdot[\text{Fläche}]}$$

Wegen der zentralen Bedeutung der Bestandsgrößen in den empirischen Wissenschafen sei hier die Vermutung ausgesprochen, daß alle Eigenschaftsgrößen aus Bestandsgrößen ableitbar sind.

Daraus würde resultieren, daß Eigenschaftsgrößen stets zu den abhängigen Variablen zählen, während Bestandsgrößen allein den Zustand bestimmen.

Mit wenigen Ausnahmen wurde diese Vermutung durch Modelle der unterschiedlichsten Fachrichtungen bestätigt. In den wenigen Fällen, in denen dies nicht zutrifft, erscheinen gar keine Bestandsgrößen im Modell. Es liegt daher der Verdacht nahe, daß diejenigen Eigenschaftsgrößen, welche als Zustandsvariablen fungieren, lediglich Bestandsgrößen substituieren.

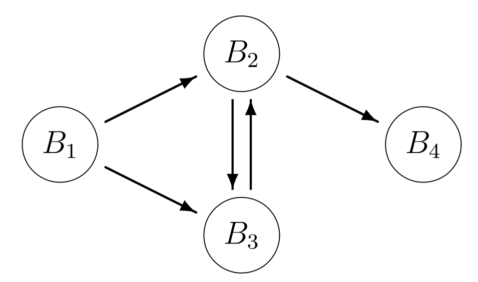
Die Vermutung wird dadurch erhärtet, daß in den Wirtschafts- und Sozialwissenschaften seit zwei Jahrzehnten eine Methode zur (graphischen) Darstellung von Modellen praktiziert wird, die nur Bestände als Zustandsgrößen kennt: System Dynamics. Auf diese Darstellungsform wird daher an späterer Stelle noch ausführlich eingegangen.

## 4.3 Beschreibung von Transporten zwischen Beständen

Es wurde bereits festgestellt, daß sich Transporte durch eine einzige Transportoperation oder durch zwei verteilte Wertzuweisungen (Wegnehmen und Hinzufügen) beschrieben werden können. In diesem Abschnitt wird diese Thematik sowohl in allgemeiner Form als auch für die Spezialfälle kontinuierlicher, diskreter und elementweiser Transport abgehandelt.

### 4.3.1 Allgemeine Betrachtungen

Gegeben seien Mengen von Beständen $B_1, B_2, \ldots$ zwischen denen Elemente ausgetauscht werden.




Mittels einer Transportfunktion $T_ij$ beschreiben wir die Transportmenge, welche zum aktuellen Zeitpunkt zwischen Bestand $B_i$ und Bestand $B_j$ ausgetauscht wird.

Je nachdem, welchen Standpunkt man einnimmt, kann der Transport als Zustandsänderung auf zwei unterschiedliche Weisen formal-sprachlich beschrieben werden.

Für einen Transport zwischen zwei einzelnen Beständen ergibt sich:

$$B_i \xrightarrow{T_{ij}} B_j$$

a) Bestandsbezogene Beschreibung
(eine Gleichung pro Bestand)

```
B_i^  :=  B_i - T_ij;
B_j^  :=  B_j + T_ij;
```

b) Transportbezogene Beschreibung
(eine Gleichung pro Transportweg)

```
(B_i -> B_j) :=  T_ij;
```

Findet auch ein Transport in der Gegenrichtung statt, geht die zusätzliche Transportmenge $T_{ji}$ in die bestandsbezogene Beschreibung als weiterer Term und in die transportbezogene Beschreibung als weitere Transportgleichung ein.

$B_i$ $\overset{T_{ij}}{\underset{T_{ji}}{\rightleftarrows}}$ $B_j$

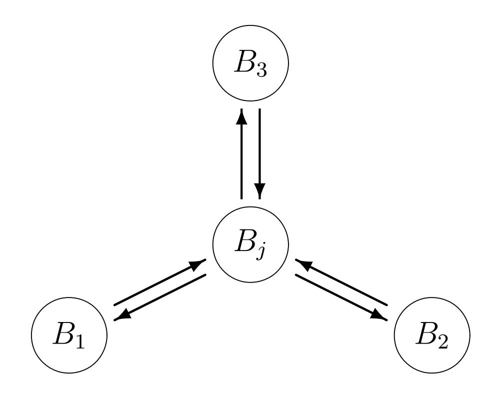
a) Bestandsbezogene Beschreibung
(eine Gleichung pro Bestand)

```
B_i^  :=  B_i - T_ij + T_ji;
B_j^  :=  B_j + T_ij - T_ji;
```

b) Transportbezogene Beschreibung
(eine Gleichung pro Transportweg)

```
(B_i -> B_j) :=  T_ij;
(B_j -> B_i) :=  T_ji;
```

Finden mehrere Transporte von und zu einem Bestand statt, dann sind die Modellbeziehungen entsprechend zu ergänzen.




a) Bestandsbezogene Beschreibung
(eine Gleichung pro Bestand)

```
B_j^  :=   B_j  +  SUM_i (T_ij)  -  SUM_i (T_ji);      i  IN  {1,2,3,..}

                   zufliessende     abfliessende
                          Transportmengen
```

b) Transportbezogene Beschreibung
(eine Gleichung pro Transportweg)

```
(B_1 -> B_j) :=  T_1j;      (B_j -> B_1) :=  T_j1;
(B_2 -> B_j) :=  T_2j;      (B_j -> B_2) :=  T_j2;
           ....                        ....
(B_i -> B_j) :=  T_ij;      (B_j -> B_i) :=  T_ji;

     zufliessende                 abfliessende
                  Transportmengen
```

Die Äquivalenz zwischen bestands- und transportbezogener Beschreibung ist uns bereits von Transporten mit individuellen Elementen bekannt.

Mit den folgenden Ausführungen präzisieren wir die transportbezogene Beschreibung auch für diskrete und kontinuierliche Transporte.

### 4.3.2 Kontinuierliche Transporte

Die Transportfunktion gibt in diesem Fall diejenige Transportmenge an, die pro Zeiteinheit zwischen Bestand $B_i$ und Bestand $B_j$ fließt (Differentialquotient). Wir bezeichnen diese Größe auch als Transportrate $r_{ij}$.

$$\text{Einheit der Transportrate} \qquad [r] \quad = \quad \frac{[B]}{[t]}$$

Eine Unterscheidung der Transportrichtung wird bei kontinuierlichen Transporten in der Regel nicht vorgenommen. Die Transportrate $r_i j$ steht demnach sowohl für den Transport von $i$ nach $j$ als auch für den Transport von $j$ nach $i$. Dies bedeutet, daß die Transportraten sowohl positive wie negative Werte annehmen können.

a) bestandsbezogene Beschreibung

Für jeden Bestand $B_j$ wird eine Gleichung angegeben, welche alle in den Bestand hineinfließenden Ströme berücksichtigt. Diese können auch negatives Vorzeichen besitzen.

```
RATE (B_j) =  B_j' :=  SUM_k(r_ij);

                zufliessende (bzw. abfliessende)
                           Stroeme
```

b) transportbezogene Beschreibung

Für jeden Transportweg zwischen zwei Beständen wird eine Gleichung folgender Form angegeben:

```
RATE (B_i -> B_j)  =  (B_i -> B_j)'  :=  r_ij (W(t));
```

### 4.3.3 Diskrete Transporte

Die Transportfunktion $T_{ij}$ ist in diesem Fall eine diskrete Funktion und liefert eine positive, ganze Zahl, die nur dann von Null verschieden ist, wenn ein Transport von Bestand $B_i$ nach Bestand $B_j$ stattfindet. Der Wert der Funktion zeigt an, wieviele Einheiten zum aktuellen Zeitpunkt den Bestand wechseln.

$$T_{ij} \quad := \quad \begin{cases} \dots & \text{falls Transportbedingung erfüllt} \\ 0 & \text{sonst} \end{cases}$$

Da der Hin- und Rücktransport zwischen zwei Beständen nur selten zeitlich zusammenfällt, ist für jede Transportrichtung eine eigene Transportfunktion anzugeben.




a) bestandsbezogene Beschreibung

```
B_j^ :=  B_j  +  SUM_i (T_ij)  -  SUM_i (T_ji);

                 zufliessende     abfliessende
                           Stueckzahlen
```

oder in Anlehnung an eine Differenzengleichung:

```
DELTA B_j :=  SUM_i (T_ij)  -  SUM_i (T_ij);
```

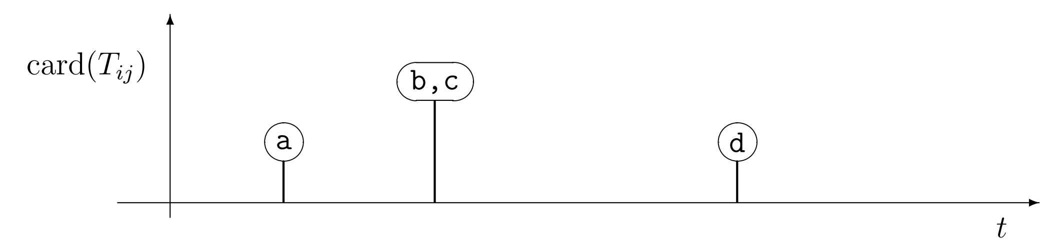
b) transportbezogene Beschreibung

```
(B_i -> B_j)  :=  T_ij;
```

### 4.3.4 Mengenweise und elementweise Transporte

In diesem Fall zeigt die Transportfunktion $T_{ij}$ an, welche Teilmenge von Elementen aus dem Bestand $B_i$ zum aktuellen Zeitpunkt zum Bestand $B_j$ überwechselt. Findet gerade kein Transport statt, umfaßt sie die leere Menge.

$$T_{ij} \quad := \quad \begin{cases} \ldots & \text{falls Transportbedingung erfüllt} \\ \emptyset & \text{sonst} \end{cases}$$




a) bestandsbezogene Beschreibung

```
                                                     SUM, + : Vereinigung
B_j^ :=  B_j  +  SUM_i(T_ij) - SUM_i(T_ji);
                                                         - : Mengendifferenz
                 zufliessende  abfliessende
                          Elemente
```

oder in Anlehnung an eine Differenzengleichung:

```
DELTA B_j :=  SUM_i (T_ij)  -  SUM_i (T_ij);
```

b) transportbezogene Beschreibung

```
(B_i -> B_j)  :=  T_ij;
```

Da $T_{ij}$ aus $B_i$ entnommen sein muß, ist für den Fall des mengenweisen Transports die kürzere Schreibweise

```
T_ij -> B_j;
```

vorteilhaft.

Bei Verwendung des Allquantors wurde als weitere Möglichkeit bereits vorgestellt, Transporte elementweise zu formulieren.

```
FOREACH  Element  IN  T_ij
LET
     Element ->  B_j;
END
```

Die Vorteile dieser Schreibweise werden an späterer Stelle behandelt.

Bei der Formulierung eines mengen- bzw. elementweisen Transports können nur Elemente aus einem Bestand entnommen werden, die auch darin enthalten sind.

Wird der Transport durch Stückzahlen beschrieben, ist es durchaus möglich, daß aus einem Bestand mehr Elemente entnommen werden sollen als dieser enthält. Der Bestand würde dann eine negative Zahl annehmen. Solche Modellierungsfehler lassen sich aber leider erst zur Laufzeit feststellen.

### 4.3.5 Gemeinsamkeiten bei der Beschreibung von Transporten

Wie man sieht, benötigen wir je nachdem, ob die Transporte individuelle Elemente umfassen, diskret oder kontinuierlich sind, etwas unterschiedliche Schreibweisen. Der strukturelle Aufbau einer Transportbeschreibung ist jedoch stets derselbe:

1. (a) Bestimmen des Transportzeitpunkts
   (außer beim kontinuierlichen Transport)
   (b) Bestimmen der Transportmenge bzw. der Transportrate
2. (a) Wegnehmen der Transportmenge von Bestand $B_i$
   (b) Hinzufügen der Transportmenge zu Bestand $B_j$

Die transportbezogene Beschreibung faßt die beiden Schritte 2a und 2b in einer Beziehung zusammen. Die Transportmenge wird daher nur in einer Beziehung benötigt. Daraus ergeben sich folgende Vorteile:

- Die Schritte 1 und 2 lassen sich in einer einzigen Zeile zusammenfassen. Für die Transportmenge $T_{ij}$ wird keine eigene Variable benötigt.
- Es ist garantiert, daß Elemente, die einem Bestand entnommen werden, auch einem anderen hinzugefügt werden. Die Erhaltung der Elemente ist damit sichergestellt.
- Die Schreibweise ist kompakt und gut verständlich.

Bei der bestandsbezogenen Beschreibung wird die Transportmenge in zwei getrennten Gleichungen benötigt. Deshalb bietet es sich an, die Transportmenge an gesonderter Stelle zu bestimmen und in einer eigenen (abhängigen) Variablen festzuhalten.

Aber auch aus anderen Gründen läßt es sich nicht umgehen, eine Zwischenvariable zu verwenden. Der folgende Abschnitt klärt die Ursachen hierfür und bietet eine Lösung an, die eine explizite Bestimmung der Transportmenge vermeidet.

### 4.3.6 Bestandsbezogene Beschreibung des Transports ohne explizite Bestimmung der Transportmenge

Das Hauptproblem besteht darin, daß eine Summenbildung über abschnittsweise definierte Funktionen in geschlossener Form nicht möglich ist. Insbesondere bei diskreten und elementweisen Transporten ist die Transportmenge immer durch Fallunterscheidungen vereinbart. Bei kontinuierlichen Transporten ist dies in vielen Fällen hingegen nicht der Fall. Eine Zusammenfassung der Gleichungen ist dann problemlos möglich.

Wir betrachten hierzu das folgende Beispiel:

```
         n_12              n_23
(B_1)  ---------->  (B_2)  ---------->  (B_3)
        cond_12            cond_23
```

```
T_12 :=  n_12  WHEN  cond_12  ELSE
         0                    END

T_23 :=  n_23  WHEN  cond_13  ELSE
         0                    END

B_2^ :=  B_2 + T_12 - T_23;
```

Eine Substitition der Variablen `T_12` und `T_23` in der letzten Zeile durch einen Ausdruck ist nicht möglich, weil die Sprachsyntax keine entsprechenden Konstruktionen vorsieht.

Eine Erweiterung der Syntax für algebraische Ausdrücke allein aus diesem Grund erscheint jedoch nicht geboten.

Auch eine Fallunterscheidung für die Bestandsgleichung führt zu keiner praktikablen Lösung, weil alle Fälle, bei denen ein gleichzeitiger Transport stattfinden kann, einzeln berücksichtigt werden müssen.

Beispiel:

```
B_2^ :=  B_2 + n_12 - n_23  WHEN  cond_12 AND cond_13  ELSE
         B_2 + n_12         WHEN  cond_12              ELSE
         B_2 - n_23         WHEN  cond_13              END
```

Falls zum Bestand $B_2$ nicht zwei sondern drei Transportwege führen würden, wären insgesamt bereits sieben Fälle zu unterscheiden.

Eine Sequentialierung in dem Sinne, daß zuerst der eine und dann der andere Transport ausgeführt wird, ist aber auch nicht möglich, weil sonst die Bestände nicht mehr zeitgleich verändert werden.

Beispiel:

```
B_2^ :=  B_2 + n_12   WHEN  cond_12  ELSE
         B_2 - n_23   WHEN  cond_13  END

B_3^ :=  B_3 + n_23   WHEN  cond_13  END
```

Wenn beide Transportbedingungen gleichzeitig erfüllt sind, wird zunächst der Bestand $B_2$ um $n_{12}$ erhöht und der Bestand $B_3$ um $n_{23}$ erhöht. Erst im folgenden Zeitpunkt wird auch der Bestand $B_2$ um $n_{23}$ vermindert, vorausgesetzt die Bedingung `cond_13` ist dann noch erfüllt.

Eine vernüftige Lösung würde sich ergeben, wenn man erlaubt, daß die Bestandsgleichung in ihre Komponenten zerlegt werden darf. Im Falle eines Transports von $B_1$ nach $B_2$ und von $B_2$ nach $B_3$, würden die Wertzuweisungen

```
B_1^ :=  B_1  -  T_12;
B_2^ :=  B_2  +  T_12 - T_23;
B_3^ :=  B_3  +  T_23;
```

ersetzt durch die relativen Aussagen

```
CHANGE B_1 :=  -T_12;        CHANGE B_2 :=  T_12;
CHANGE B_2 :=  -T_23;        CHANGE B_3 :=  T_12;
```

Je zwei dieser Aussagen beschreiben einen Transport. Dadurch wird eine starke Ähnlichkeit zur transportbezogenen Beschreibung hergestellt.

Allgemein betrachtet wird die Wertzuweiung

```
B_j^ :=  B_j  +  T_1j + T_2j + ... + T_nj  - T_j1 - T_j2 - ... - T_jn
```

dabei ersetzt durch die Aussagen:

```
CHANGE B_j :=  T_1j;         CHANGE B_j :=  - T_j1;
CHANGE B_j :=  T_2j;         CHANGE B_j :=  - T_j2;
...
CHANGE B_j :=  T_nj;         CHANGE B_j :=  - T_jn;

zufliessende  Elemente       abfliessende Elemente
```

Die Summenbildung wird implizit über alle CHANGE–Anweisungen ausgeführt, die den gleichen Bestand betreffen.

```
B_j ^ :=  B_j  +  SUM ( CHANGE B_j );
```

Bei Einführung dieses Konstrukts kann auf die explizite Berechnung der Transportmenge verzichtet werden und die Modellbechreibung gewinnt sogar etwas an Übersicht.

Beispiel:

```
CHANGE B_1 := -n_12  WHEN  cond_12  END
CHANGE B_2 :=  n_12  WHEN  cond_12  END

CHANGE B_2 := -n_23  WHEN  cond_23  END
CHANGE B_3 :=  n_23  WHEN  cond_23  END
```

Zum Vergleich sei abschließend auch die äquivalente, transportbezogene Beschreibung angegeben.

Beispiel:

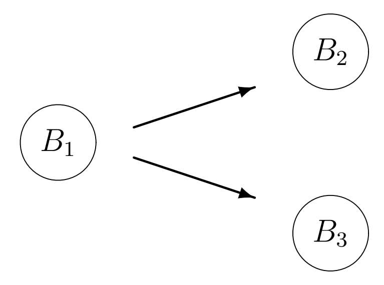
```
(B_1 -> B_2) :=  n_12  WHEN  cond_12  END
(B_2 -> B_3) :=  n_23  WHEN  cond_13  END
```

Inwieweit durch die unterschiedlichen Beschreibungsformen die Eindeutigkeit sichergestellt ist, wird im nächsten Abschnitt behandelt.

## 4.4 Diskussion der Eindeutigkeit am Problem der Verteilung

Die Beschreibung der Transportvorgänge in einem Modell ist dann eindeutig, wenn nicht der Fall eintreten kann, daß ein Element an zwei verschiedene Orte transportiert werden soll.

Probleme mit der Eindeutigkeit können daher immer dann auftreten, wenn Elemente aus einem Bestand auf zwei oder mehrere Bestände verteilt werden.




Zum Studium der möglichen Konfliktfälle und zum Vergleich der verschiedenen Gegebenheiten wollen wir wiederum zwischen kontinuierlichen, diskreten, mengenweisen und elementweisen Transporten unterscheiden.

### 4.4.1 Kontinuierlicher Transport

1. Bestimmen der Transportraten

```
r_12 :=  ... ;
r_13 :=  ... ;
```

2. Beschreibung des Zustandsänderung

```
a) bestandsbezogen         B_1' :=  - r_12 - r_13;
                           B_2' :=  r_12;
                           B_3' :=  r_13;

b) transportbezogen        (B_1 -> B_2)' :=  r_12;
                           (B_1 -> B_3)' :=  r_13;
```

In beiden Fällen sind die Differentialquotienten eindeutig definiert.

Wenn – wie in den meisten Fällen – zur Berechnung der Transportraten keine Fallunterscheidungen nötig sind, brauchen diese nicht gesondert berechnet werden, sondern können bei der Angabe des Differentialquotienten bestimmt werden.

## 4.4.2 Diskreter Transport

1. Bestimmen der Transportfunktionen

```
IF cond_12  LET  T_12 := n_12;  END
      ELSE  LET  T_12 :=  0  ;  END

IF cond_13  LET  T_13 := n_13;  END
      ELSE  LET  T_13 :=  0  ;  END
```

2. Beschreibung des Zustandsübergangs

```
a) bestandsbezogen       B_1^ :=  B_1 - T_12 - T_13;
                         B_2^ :=  B_2 + T_12;
                         B_3^ :=  B_3 + T_13;
   durch Angabe der
   Differenzen:          DELTA B_1 :=  -T_12 - T_13;
                         DELTA B_2 :=   T_12;
                         DELTA B_3 :=   T_13;

   komponentenweise:     CHANGE B_1 :=  -T_12;     CHANGE B_2 :=  T_12;
                         CHANGE B_1 :=  -T_13;     CHANGE B_3 :=  T_13;


b) transportbezogen      B_1 -> B_2  :=  T_12;
                         B_1 -> B_3  :=  T_13;
```

Die Eindeutigkeit ist auch hier soweit sichergestellt. Mehrdeutige Formulierung, wie sie etwa das folgende Beispiel zeigt, können - wie bereits besprochen - vom Compiler erkannt und angemahnt werden.

Beispiel:

```
IF  cond_12  LET  B_1^ :=  B_1 - n_12;
                  B_2^ :=  B_2 + n_12;  END

IF  cond_23  LET  B_2^ :=  B_2 - n_23;
                  B_3^ :=  B_3 + n_23;  END
```

Der Bestand `B_2` ist mehrdeutig bestimmt, wenn die Transportbedingungen `cond_12` und `cond_23` gleichzeitig erfüllt sind.

## 4.4.3 Mengenweiser Transport

Wir betrachten den Fall, daß beim Eintritt einer Bedingung eine Teilmenge aus dem Bestand $B_1$ nach $B_2$ und beim Eintritt einer anderen Bedingung eine andere Teilmenge aus $B_1$ nach $B_3$ gebracht werden soll.

Beispielsweise sollen bei leerem Bestand $B_2$ bis zu fünf Aufträge des Typs `'A'` (Bedingung `cond_12`) nach $B_2$ und bei leerem Bestand $B_3$ bis zu zehn Aufträge des Typs `'B'` (Bedingung `cond_13`) nach $B_3$ gebracht werden.



1. Bestimmen der Transportfunktion

```
T_12 :=  { J IN B_1   |  cond_12 }: Job {1..n_12};
T_13 :=  { J IN B_1   |  cond_13 }: Job {1..n_13};
```

2. Beschreibung des Zustandsübergangs

```
a) bestandsbezogen        B_1^ :=  B_1 - T_12 - T_13;
                          B_2^ :=  B_2 + T_12;
                          B_3^ :=  B_3 + T_13;
   durch Angabe der
   Differenzen:           DELTA B_1 :=  -T_12 - T_13;
                          DELTA B_2 :=   T_12;
                          DELTA B_3 :=   T_13;

   komponentenweise:      CHANGE B_1 :=  -T_12;    CHANGE B_2 :=  T_12;
                          CHANGE B_1 :=  -T_13;    CHANGE B_3 :=  T_13;


b) transportbezogen       T_12 -> B_2;
                          T_13 -> B_3;

   elementweise           FOREACH  Job  IN  B_1
                          LET
                              IF  Job  IN  T_12  LET  Job -> B_2;  END
                              IF  Job  IN  T_13  LET  Job -> B_3;  END
                          END
```

Die Beschreibung entspricht der beim diskreten Transport mit dem Unterschied, daß die Bestände Mengen statt Anzahlen ausdrücken. daß Mengenvariablen anstelle von Wertvariablen beteiligt sind. Statt Summe und Differenz stehen hier die Operationen Mengenvereinigung und Mengendifferenz.

Aber es taucht hier ein Problem auf, das beim diskreten Transport nicht besteht. Die Bestimmung der Transportmengen auf die obige Art gewährleistet nicht, daß die Transportmengen $T_{12}$ und $T_{13}$ disjunkt sind. Es ist nämlich nicht ohne weiteres erkenntlich, ob sich die Bedingungen `cond_12` und `cond_13` gegenseitig ausschließen. Dies ist aber eine notwendige Voraussetzung dafür, daß für jedes Element aus $B_1$ eindeutig bestimmt ist, ob es nach $B_2$ oder $B_3$ transportiert wird.

Bei den folgenden Transportfunktionen wäre dies beispielsweise erfüllt, ist aber im allgemeinen Fall vom Compiler nicht nachzuweisen.

```
T_12 :=  { J IN B_1  |  J.Typ = 'A'  AND  cond_12 } : Job {1..n_12};
T_13 :=  { J IN B_1  |  J.Typ = 'B'  AND  cond_23 } : Job {1..n_13};
```

Dieses Problem läßt sich

a) durch ein Sprachkonstrukt, dessen Syntax die Disjunktheit der Transportmengen garantiert oder

b) durch einen elementweisen Transport mobiler Elemente

lösen.

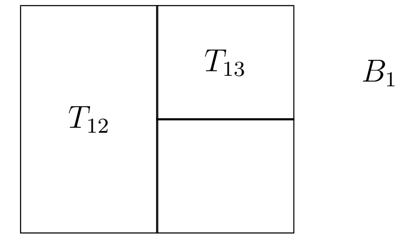
### 4.4.4 Das DEVIDE-Konstrukt

Angeregt wird ein Sprachkonstrukt, das ähnlich dem IF-THEN-ELSE sukzessive einen angegebenen Bestand unterteilt.

Dieses Konstrukt, hier als DEVIDE-Konstrukt bezeichnet, könnte etwa folgendermaßen aussehen:

```
DEVIDE  B_1
  INTO  T_12 :=  { Job | cond_12 }: Job{1..n_12};
  REMAINING SET
  INTO  T_13 :=  { Job | cond_13 }: Job{1..n_13};
END
```

Aus $B_1$ wird zunächst die Teilmenge $T_{12}$ gebildet und aus der verbliebenen Menge $B_1 - T_{12}$ wird die Teilmenge $T_{13}$ ausgegrenzt. Elemente aus den Eigenschaftsmengen werden auf diese Art eindeutig der Teilmenge $T_{12}$ oder $T_{13}$ zugeordnet.



Die Eindeutigkeit beim Tranportvorgang ist hergestellt, wenn Transportmengen, welche Teilmengen des gleichen Bestandes sind, unter Zuhilfenahme des DEVIDE-Kontrukts ermittelt worden sind. Das ist wiederum vom Compiler abprüfbar.

Das DEVIDE-Konstrukt impliziert eine Prioritätenregelung wie wir sie in ähnlicher Form vom IF-Konstrukt kennen: Ein Element, das in beiden Eigenschaftsmengen enthalten ist, wird bevorzugt der ersten Transportmenge zugeschlagen.

Will man die Eindeutigkeit beim Transport herstellen, ohne die Sprache um das DEVIDE-Konstrukt zu ergänzen, darf in Transportoperationen nur die Verwendung einzelner Elemente zugelassen sein, d.h. eine Angabe ganzer Mengen darf nicht ermöglicht werden.

### 4.4.5 Elementweiser Transport

Führt man den Transport elementweise durch, läßt die Syntax ebenfalls keinen Eindeutigkeits-Konflikt zu. Ein IF-THEN-ELSE-Konstrukt sorgt innerhalb des Allquantors für die Eindeutigkeit beim Verteilen der Elemente.

```
FOREACH  TJob  IN  B_1
LET
    IF     TJob  IN  T_12
    LET
           TJob -> B_2;
    END
    ELSIF  TJob  IN  T_23
    LET
           TJob -> B_3;
    END
END
```

Die Mengen $T_{12}$ und $T_{23}$ entsprechen nun allerdings u.U. nicht mehr den tatsächlichen Transportmengen. Deshalb kann es ratsam sein, auf eine gesonderte Bestimmung der Transportmengen zu verzichten.

```
FOREACH  TJob  IN  B_1
LET
    IF     TJob  IN  { Job IN B_1 | condition_12 }: Job{1..n_12}
    LET
           TJob -> B_2;
    END
    ELSIF  TJob  IN  { Job IN B_1 | condition_13 }: Job{1..n_13}
    LET
           TJob -> B_3;
    END
END
```

Setzt man anstelle der Transportoperation

```
TJob -> B_i
```

die beiden Aussagen

```
CHANGE B_1 :=  -TJob;     CHANGE B_i :=  TJob;
```

ein, läßt sich die Eindeutigkeit auch für die bestandsbezogene Beschreibung herstellen.

Für den häufigen Fall, daß immer nur ein Element transportiert werden soll, läßt sich die Verteilung aus einem Bestand wesentlich kürzer und ohne Bestimmung einer Eigenschaftsmenge durchführen.

Beispiel:

```
FOREACH  TJob  IN  B_1: Job{1}
LET
    IF     cond_12
    LET
           TJob -> B_2;
    END
    ELSIF  cond_13
    LET
           TJob -> B_3;
    END
END
```

Die bisherigen Sprachkonstrukte reichen demnach durchaus aus, um das Problem des eindeutigen Zugriffs bei Verteilung zu lösen.

# 4.5 Modularisierung

Die hierarchische Modularisierbarkeit von Modellen wurde als eines der wichtigsten Ziele ganannt. In diesem Abschnitt soll nun untersucht werden, welche prinzipiellen Möglichkeiten bestehen, um Modelle bestandsbezogen oder transportbezogen zu zerlegen.

Auch bei modular zerlegten Modellen durch eine geschickte Sprachsyntax eine Überprüfung der semantischen Korrektheit durch den Compiler erfolgen können.

Zu den bisher genannten Anforderungen kommt nun noch die Autonomie einer Komponente, die in einem eigenen Abschnitt diskutiert wird.

Anforderungen, die im nicht zerlegten Modell verletzt werden, lassen sich natürlich im zerlegten Modell erst recht nicht erfüllen. Hierzu zählt insbesondere die Erhaltung der Elemente, die von der bestandsbezogenen Beschreibung nicht sichergestellt werden kann.

Die Gewährleistung der Eindeutigkeit wird auch weiterhin Ziel der Bemühungen sein. Ebenso die Möglichkeit, Zusammenhänge kompakt zum Ausdruck zu bringen. Für die Verständlichkeit eines modular aufgebauten Modells ist vor allen Dingen wichtig, daß der Transportvorgang und die Transportrichtung zwischen zwei Komponenten ersichtlich ist.

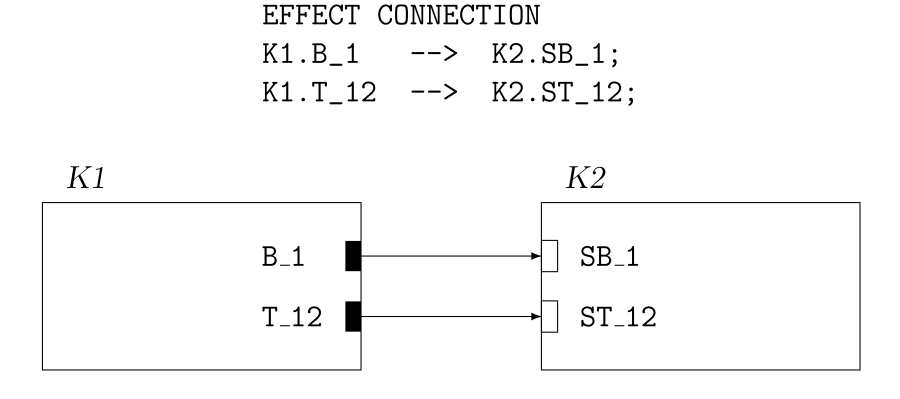
## 4.5.1 Modularisierung mit bestandsbezogener Beschreibung

Die Modularisierung bei bestandsbezogener Beschreibung wird ebenso vorgenommen wie die Modularisierung mit Eigenschaftsgrößen: Jeder Bestand und dessen Definitionsgleichung wird einer (beliebigen) Komponente zugeordnet.

Wie wir bereits wissen, ermöglicht diese Vorgehensweise einen hierarchischen Modellaufbau. Auch können die Komponenten als autonom angesehen werden, da der Zeitverlauf einer (internen) Variablen nur durch eine Definitionsgleichung festgelegt ist und diese sich in der gleichen Komponente befindet. In diesem Abschnitt soll untersucht werden, inwieweit auch die anderen Anforderungen an eine Modellbeschreibungssprache bei bestandsbezogener Transportbeschreibung erfüllt werden.

In diesem Abschnitt wird die modulare Modellzerlegung, wie sie im Abschnitt 4.1 für Transportvorgänge angedeutet wurde, verallgemeinert. Hierzu sind neben den Beständen (bezeichnet mit `B`) abhängige (Mengen-) Variablen notwendig, um die Transportmenge (bezeichnet mit `T`) ausdrücken und an andere Komponenten weitergeben zu können.

Weiterhin sind Sensorvariablen für Mengen erforderlich, um Einblick in Bestände (bezeichnet mit `SB`) und Transportmengen (bezeichnet mit `ST`) fremder Komponenten nehmen zu können.



Die Typen von Variablen sind daher die gleichen wie bei den Eigenschaftsvariablen. Anstatt mit einer Wertemenge (`INT, REAL, ...`) operieren sie jedoch auch mit einer Menge von mobilen Elementen (`SET OF mobile_component`).

Wir wollen nun die bestehenden Möglichkeiten diskutieren, um mit der bestandsbezogenen Beschreibung Transportmodelle zu zerlegen, und deren Vor- und Nachteile aufzeigen.

Hierzu betrachten wir zwei Bestände, zwischen denen bidirektionale Transporte stattfinden.

```
                T_ij
   (B_i)   ------------>   (B_j)
           <------------
                T_ji
```

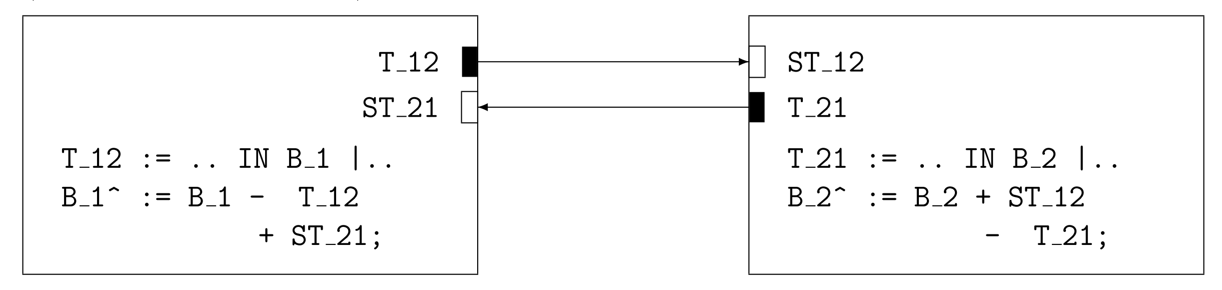
```
B_1^ :=  B_1 - T_12              B_2^ := B_2 + T_12
             + T_21;                         - T_21;
```

Bei dem Versuch, die Bestände `B_1` und `B_2` und deren Zustandsüberführungsfunktionen in zwei verschiendene Komponenten zu legen, fällt es sofort als störend auf, daß die Transportmengen `T_12` und `T_21` in beiden Komponenten benötigt werden. Die Transportmengen müssen daher entweder

- in einer Komponente (Quellkomponente oder Zielkomponente) ermittelt und an die andere Komponente weitergegeben werden
- in beiden Komponenten ermittelt werden
- in einer gesonderten Komponente bestimmt werden.

Dem Anwender stehen hiermit vier Möglichkeiten zur Auswahl, den Transport zwischen zwei Komponenten zu beschreiben.

1. Bestimmung der Transportmenge in der Quellkomponente
   (Bringen von Elementen)



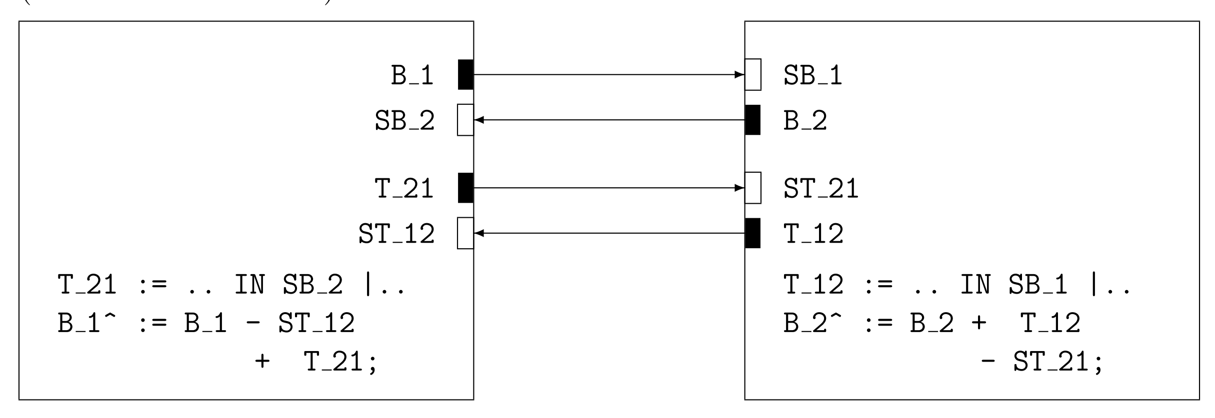

   Die Menge der zu transportierenden Elemente wird an die andere Komponente übermittelt.

   Gegenüber den später angeführten Möglichkeiten ergeben sich in diesem Fall nur Vorteile:

   - Eindeutigkeit
     läßt sich gewährleisten.
   - Kompaktheit
     Die Transportmenge wird nur einmal bestimmt.

- Verständlichkeit
  Die Verbindungen geben Aufschluß über einen Transportpfad.
  Die Richtung der Verbindung stimmt mit der Transportrichtung überein.

2. Bestimmung der Transportmenge in der Zielkomponente
   (Holen von Elementen)

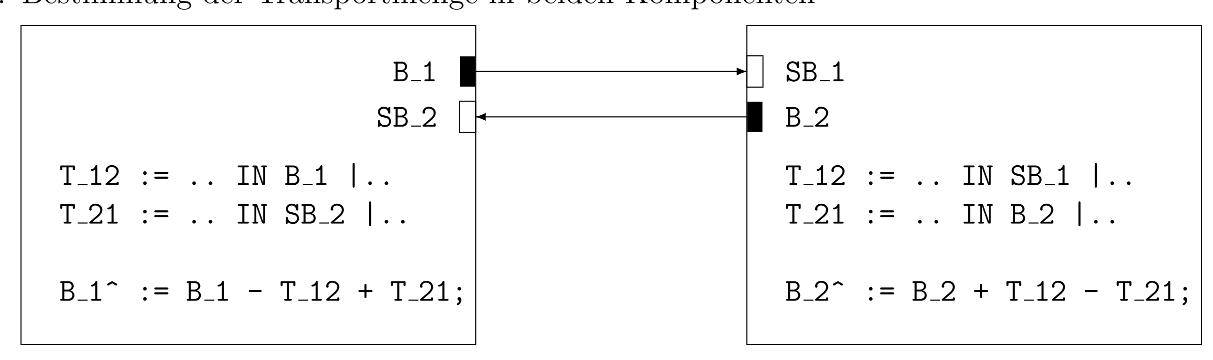



Neben der Menge der zu transportierenden Elemente sind an die andere Komponente auch die beteiligten Bestände zu übermitteln.

In diesem Fall ergeben sich nur Nachteile:

- Eindeutigkeit
  läßt sich für den Transport individueller Elemente nicht gewährleisten. Ein Element kann von der eigenen ebenso wie von einer fremden Komponente bewegt werden.
- Kompaktheit
  Es sind zusätzliche Sensorvariablen und Verbindungen erforderlich.
- Verständlichkeit
  Die Verbindungen zur Weitergabe der Transportmengen weisen entgegen der Transportrichtung.

3. Bestimmung der Transportmenge in beiden Komponenten



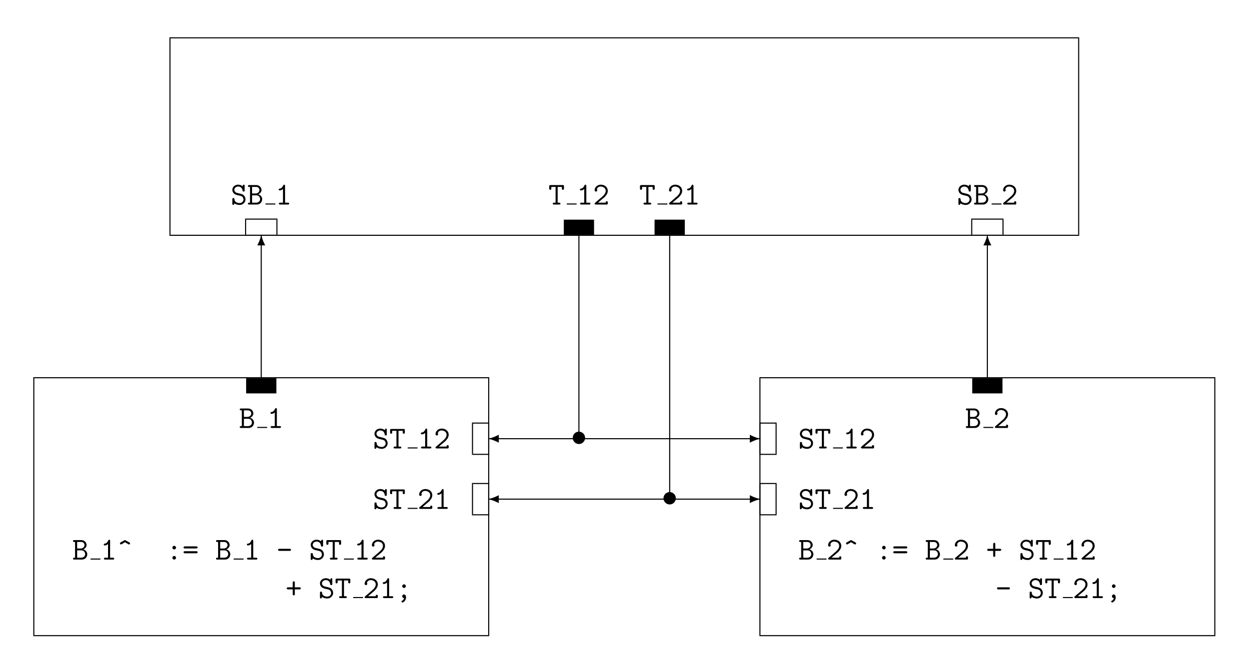

Vorteile:

- Eindeutigkeit
  läßt sich gewährleisten.
- Symmetrie
  Eine gleichartige Beschreibung in beiden Komponenten ist vorteilhaft für Anwendung des Klassenkonzepts.

Nachteile:

- ○ Kompaktheit und Fehleranfälligkeit
  Transportmenge wird zweifach bestimmt und muß identisch sein.
- ○ Verständlichkeit
  Die Verbindungen zwischen den Komponenten lassen nicht auf einen Transport schließen.
  In der Regel erhöhter Kommunikationsaufwand, da in beiden Komponenten die Information zur Berechnung der Transportmengen vorliegen muß.

4. Bestimmung der Transportmenge in einer gesonderten Komponente




Alle zur Bestimmung der Transportmengen benötigten Informationen werden der gesonderten Komponente zur Verfügung gestellt. Die ermittelten Transportmengen werden dann wiederum an die Komponenten weitergegeben.

Vorteile:

- • symmetrische Beschreibung
  (vorteilhaft für Anwendung des Klassenkonzepts)

Nachteile:

- ○ Eindeutigkeit
  läßt sich für den Transport individueller Elemente nicht gewährleisten.
- ○ hoher Komunikationsaufwand
  Alle Information zur Berechnung der Transportmengen müssen der Zwischenkomponente zur Verfügung gestellt werden.
- ○ Verständlichkeit
  Die Verbindungen zwischen den Komponenten lassen nicht auf die Transportrichtung schließen.

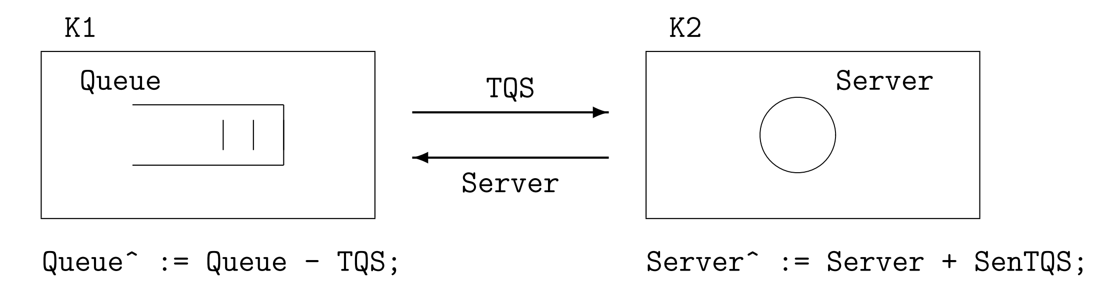
Wie wir feststellen, ist nur bei Variante 1 für den Transport individueller Elemente die Eindeutigkeit gesichert und der Transportvorgang auf der höheren Ebene zufriedenstellend zu erkennen.

Ein Beispiel soll dies weiter verdeutlichen. Außerdem ist zu ersehen, daß neben der Übergabe der Transportmenge in der Regel noch weitere Kommunikation erforderlich ist, um die Transportmenge überhaupt bestimmten zu können.

<u>Beispiel:</u> Transport von Warteschlange Queue nach Bedieneinheit Server

Die Bestände befinden sich in verschiedenen Komponenten.
`TQS:` Transportmenge Queue-Server
`SenTQS:` Sensorvariable für Transportmenge

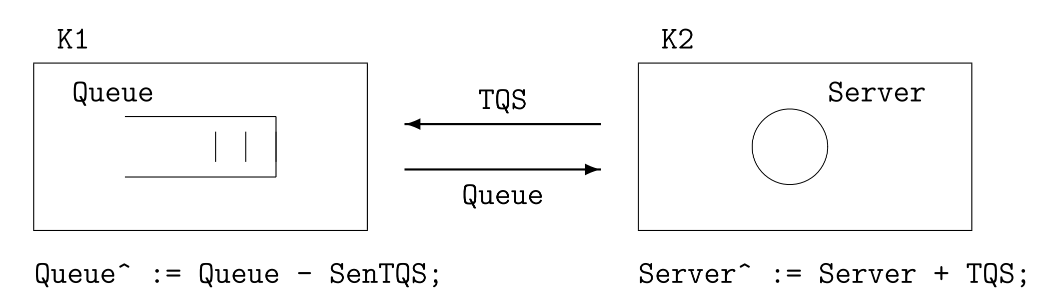
1. Bestimmung der Transportmenge in der Quellkomponente




Bestimmung der Transportmenge in `K1`:

```
IF  EMPTY SenServer  LET  TQS := Queue:Job{1};  END
                ELSE  LET  TQS := {};            END
```

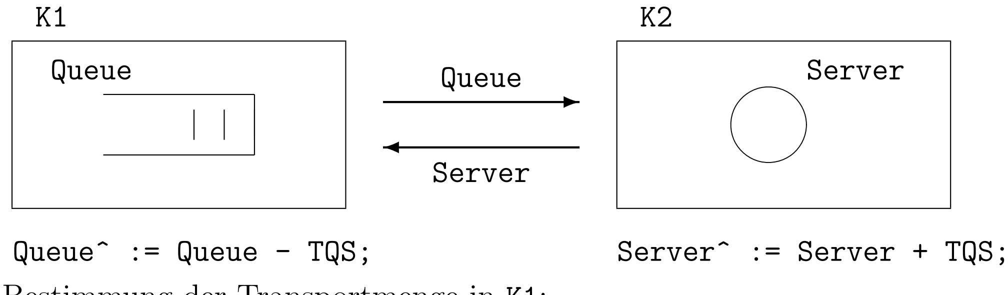
2. Bestimmung der Transportmenge in der Zielkomponente




Bestimmung der Transportmenge in `K2`:

```
IF  EMPTY Server  LET  TQS := SenQueue:Job{1};  END
            ELSE  LET  TQS := {};               END
```

3. Bestimmung der Transportmenge in beiden Komponenten




Bestimmung der Transportmenge in K1:

```
IF  EMPTY SenServer  LET  TQS := Queue:Job{1};  END
               ELSE  LET  TQS := {};            END
```

Bestimmung der Transportmenge in K2:

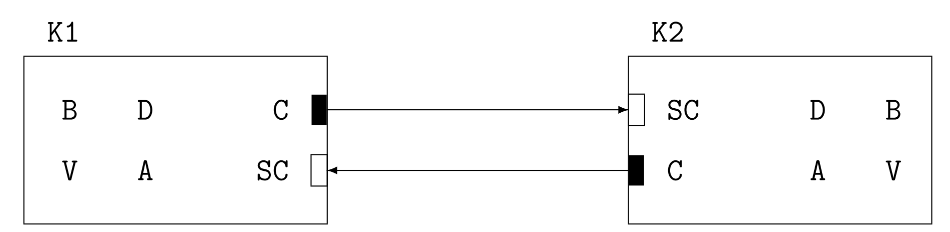
```
IF  EMPTY Server  LET  TQS := SenQueue:Job{1};  END
            ELSE  LET  TQS := {};               END
```

Das Beispiel macht aber auch deutlich, daß die in den allgemeinen Überlegungen genannten Vor- und Nachteile gar nicht so gravierend ins Gewicht fallen müssen. Eine Einschränkung des Anwenders auf die erste Variante erscheint daher nicht geboten und wäre ohnedies nur schwer realisierbar.

Für bestimmte Anwendungsfälle, insbesondere beim nicht-individuellen Transport, gestatten gerade die Varianten 3 und 4 wegen ihrer Symmetrie durchaus eine vorteilhafte Modellzerlegung. Da das Klassenkonzept eingesetzt werden kann, werden die anderen Nachteile wieder wettgemacht.

Beispiel: Diffusion durch eine Membran

Jede Komponente repräsentiert ein Behältnis. Dazwischen liegt eine teildurchlässige Membran der Fläche A. Die Diffusionskonstante ist D. Der Diffusionsstrom (Bestand/Zeit) ist dem Konzentrationsgefälle proportional. Die Konzentration errechnet sich aus Bestand pro Volumen.




Modellgleichungen in beiden Komponenten:

```
C  :=  B / V;                # Konzentration
B' :=  D/A * (SC - C);       # Aenderung des Bestandes
```

Diskussion:

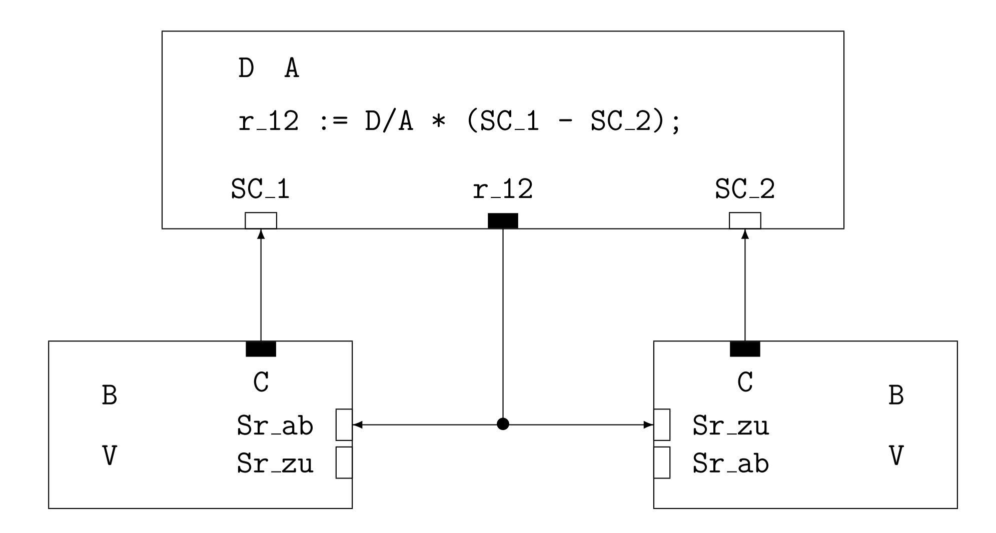
Die Kompaktheit der Beschreibung wird in erster Linie dadurch erzielt, daß keine Flußrichtung unterschieden wird. Transporte in Gegenrichtung werden einfach durch

negative Zahlenwerte ausgedrückt, eine Verfahrensweise wie sie bei Mengen aus individuellen Elementen selbstverständlich nicht möglich ist.

Trotz der eleganten, symmetrischen Beschreibung wird jedoch der Transport zwischen den Komponenten nicht deutlich.

Als nachteilig fällt auch auf, daß die Trennschicht mit Fläche `A` und Diffusionskoeffizient `D` im System nur einmal vorhanden ist, im Modell die Parameter jedoch in beiden Komponenten deklariert werden.

Abhilfe in diesen Punkten schafft eine Lösung nach Variante 4. Hier bildet die Trennschicht eine eigene Komponente.




Modellgleichungen in beiden Komponenten:

```
C :=  B / V;                  # Konzentration

B' :=  Sr_ab - Sr_zu;         # Aenderung des Bestandes
```

Jede Komponente enthält die für sie relevanten Variablen.

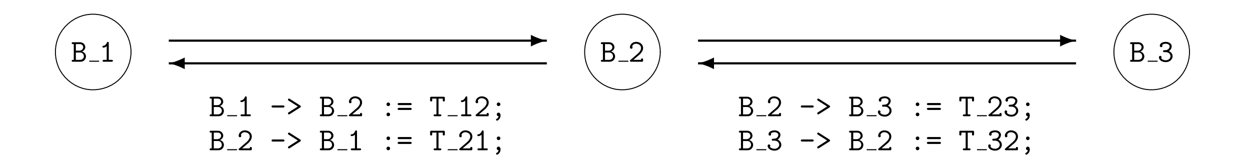
Jede Komponente besitzt durch die Sensorvariablen `Sr_ab` und `Sr_zu` die Möglichkeit ihren Bestand zu vergrößern oder zu verkleinern. Ob eine Veränderung des Bestands zu erfolgen hat, bestimmt die Membran als dazwischenliegende Komponente. Durch das Herstellen der passenden Verbindungen wirkt eine veranlaßte Änderung der Bestände für die eine Komponente positiv und für die andere Komponente negativ aus.

Dieses Beispiel sollte deutlich machen, daß die angeführten Anforderungen zu schwache Kriterien sind, um von vornherein die eine oder andere Lösung ausscheiden zu lassen. Die Anforderungen der Praxis sind oft höher zu bewerten. Es zeigt, daß für eine komponentenweise Zerlegung dem Anwender alle Möglichkeiten offenstehen sollten. Damit muß man sich allerdings im klaren sein, daß sich die Korrektheit des Modells bei einer bestandsbezogenen Transportbeschreibung nicht durch semantische Prüfungen nicht sicherstellen läßt.

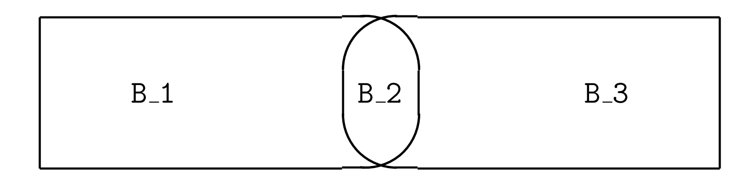
### 4.5.2 Modularisierung mit transportbezogener Beschreibung

In diesem Fall sollen nicht die Bestände mit ihren Definitionsgleichungen auf verschiedene Komponenten verteilt werden, sondern die Transportgleichungen. Welche Konsequenzen dies nach sich zieht, wird im folgenden erörtert.

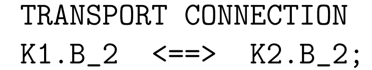
Wir betrachten hierzu drei Bestände `B_1`, `B_2`, `B_3` mit bidirektionalen Transporten zwischen `B_1` und `B_2` sowie `B_2` und `B_3`.



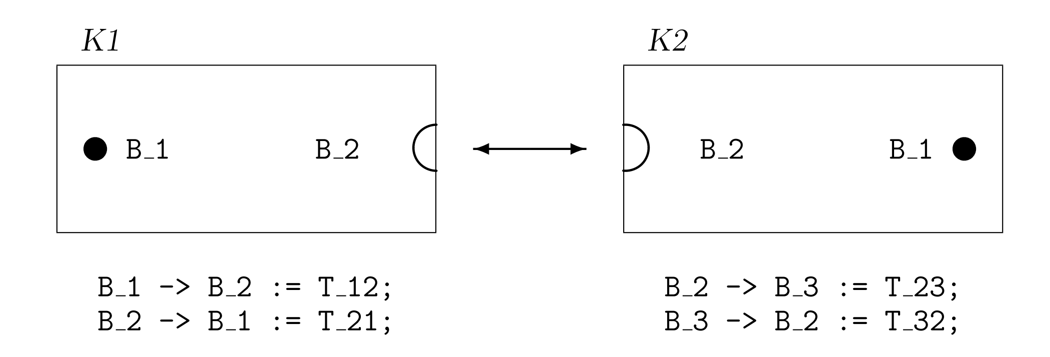

Eine komponentenweise Zerlegung nach Transportgleichungen bedingt, daß der Bestand `B_2` in beiden Komponenten auftritt. Dabei handelt es sich jedoch nicht nur um einen lesenden Zugriff, d.h. um die reine Beschaffung von Information aus einer fremden Komponente. Vielmehr können beide Komponenten auf den Bestand `B_2` gleichberechtigt zugreifen. Die beiden Komponenten teilen sich gewissermaßen den Bestand `B_2`.




Für eine Modularisierung bedeutet das, daß `B_2` in beiden Komponenten zu deklarieren ist und in der höheren Komponente eine Äquivalenzierung vorgenommen werden muß. Diese Gleichsetzung mit der Möglichkeit eines Transports bezeichnen wir als Transportverbindung. Der Operator `'<==>'` deutet an, daß ein Transport in beiden Richtungen stattfinden kann.



```
TRANSPORT CONNECTION
K1.B_2  <==>  K2.B_2;
```



Zu einem Widerspruch kommt es, wenn die Initialisierung des Bestandes B_2 nicht in beiden Komponenten identisch ist. In diesem Fall ist es erforderlich, daß die Äquivalenzierung in der höheren Komponente mit einer Vorbesetzung versehen wird, welche die Vorbesetzungen der darunter liegenden Komponenten überschreibt.

```
K1.B_2  <==>  K2.B_2  :=  <initial_value>;
```

Auch eine Hierarchisierung der Modelle ist möglich und wird später noch ausführlich behandelt. Zu diesem Zweck sind Bestände lediglich unter einem eigenen Namen an die nächst höhere Komponente weiterzugeben.

```
B  <==>  K1.B_1;
```

Gegenüber der Modularisierung mit bestandsbezogenen Gleichungen weist diese Art der Modellzerlegung eine ganze Reihe von Vorteilen auf.

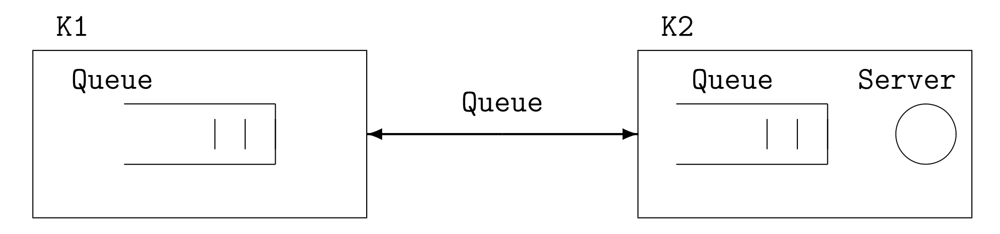
- Erhaltung
  Bei der transportbezogenen Beschreibung können keine Elemente verloren gehen oder unbeabsichtigt hinzukommen.
- Kompaktheit
  Die Beschreibung des Transports und die Bestimmung der Transportmenge können in einer einzigen Beziehung zusammengefaßt werden. Der Kommunikationsaufwand ist deutlich geringer.
- Verständlichkeit
  Transporte zwischen Komponenten sind deutlich hervorgehoben.
- Unabhängigkeit
  Ein Teilaspekt kann vollständig modelliert und getestet werden, ohne daß die Umgebung bekannt sein muß, da Transporte nur innerhalb der eigenen Komponente beschrieben werden.

Die Vorteile zeigt auch das folgende Beispiel.

Beispiel: Transport von Warteschlange Queue nach Bedieneinheit Server

Die äquivalenzierten Bestände tragen in beiden Komponenten denselben Namen.

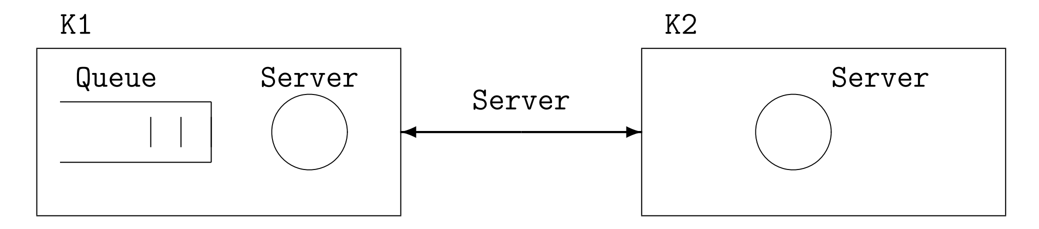
1. Äquivalenzierung der Warteschlange




Beschreibung des Transports in `K2`:

```
EMPTY Server  LET  Queue:Job{1} -> Server;  END
```

2. Äquivalenzierung der Bedieneinheit




Beschreibung des Transports in `K1`:

```
EMPTY Server  LET  Queue:Job{1} -> Server;  END
```

Ebenso wie bei der bestandsbezogenen Beschreibung treten beim Transport von individuellen Elementen Probleme mit der Eindeutigkeit auf.

Es ist möglich, daß beide Komponenten gleichzeitig verändernd auf dasselbe Element im Bestand `B_2` zugreifen. Dies kann nicht nur zur Ausführung eines Transportes, sondern auch zum Setzen eines Attributes geschehen. Damit im praktischen Betrieb kein Zugriffskonflikt auftreten kann, wäre eine genaue Abstimmung zwischen den Komponenten notwendig.

Die Ursache für einen möglichen Konflikt liegt darin, daß ein Bestand gleichberechtigt beiden Komponenten zugeschlagen wird und diese daher über die gleichen Zugriffsrechte verfügen. Keine der beiden Komponenten kann über die äquivalenzierten Bestände autonom verfügen.

Wie der nächste Abschnitt zeigt, kann durch eine teilweise Wiederherstellung der Autonomie das Eindeutigkeitsproblem gelöst werden. Für die bestandsbezogene Beschreibung hingegen konnte eine derartige Lösung nicht gefunden werden.

Für Transporte mit gleichartigen Elementen liegt bereits ohne weitere Ergänzungen eine ideale Beschreibungsform vor. In diesen Fällen stellt die Eindeutigkeit kein Problem dar. Insbesondere fällt auf, daß es keine unsymmetrischen Lösungen mehr gibt und daher jede mögliche Art der Modellierung für die Anwendung des Klassenkonzepts gut geeignet ist.

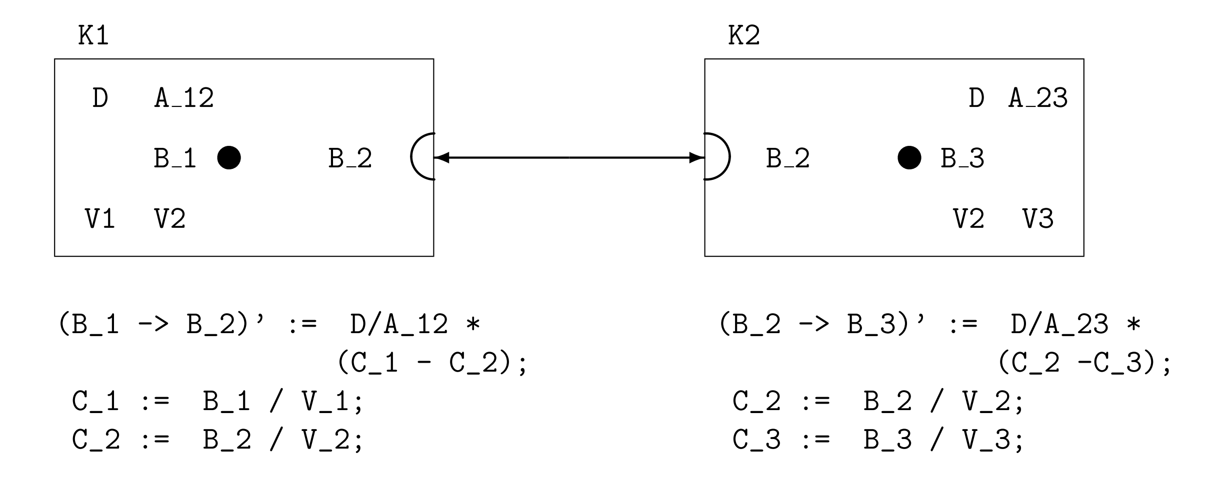
Beispiel: Diffusion

Wir betrachten drei Behältnisse mit Beständen `B_1`, `B_2`, und `B_3`, die durch zwei Membranen getrennt sind.

| B_1 | B_2 | B_3 |
|---|---|---|

Wir zerlegen das Gesamtmodell in zwei identische Komponenten. Jede Komponente beschreibt einen Diffusionsvorgang.

```
TRANSPORT CONNECTION
K1.B2  <==>  K2.B2;
```




Das Volumen `V_2` ist in beiden Komponenten zu halten, da es eine Eigenschaft des Bestandes `B_2` ist.

Eine andere Modellzerlegung, die sich mehr an den physikalischen Gegebenheiten orientiert, zeigt das folgende Beispiel. Um jeden Bestand wird eine Komponente gelegt und der Transport durch die Trennschicht in einer gesonderten Komponente modelliert. Die Modellzerlegung entspricht daher der aus dem letzten Beispiel des vorangegangenen Abschnitts.

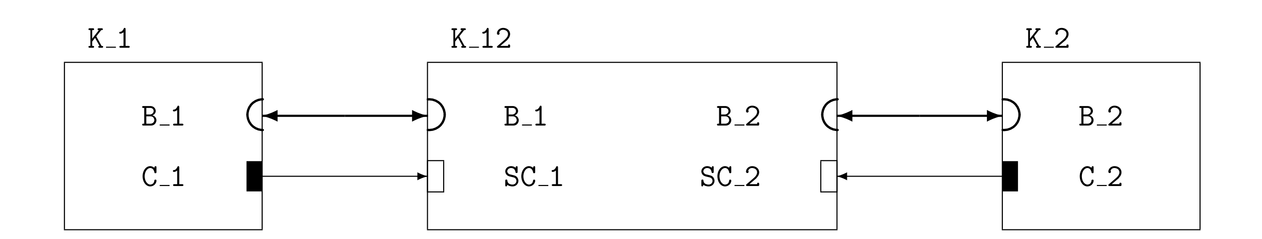
Außerdem zeigt dieses Beispiel, daß es möglich ist, ein bestehendes Modell durch Einfügen einer weiteren Komponente um einen Transportvorgang zu erweitern, ohne daß in die anderen Komponenten eingegriffen werden muß. Dies ist natürlich nur möglich, weil die Komponenten auf die Autonomie über ihre Bestände verzichten.

Beispiel: Diffusion

Wir betrachten zwei Behältnisse mit Beständen `B_1` und `B_2`, die zunächst voneinander unabhängig sind. Nun soll das Modell durch einen Diffusionsvorgang zwischen diesen beiden Behältnissen erweitert werden.

```
EFFECT CONNECTIONS
K_1.C_1  -->  K_12.SC_1;
K_2.C_2  -->  K_12.SC_2;

TRANSPORT CONNECTIONS
K_1.B_1  <==>  K_12.B_1;
K_2.B_2  <==>  K_12.B_2;
```




```
(B_1 -> B_2)' :=  D/A * (C_1 - C_2);
```

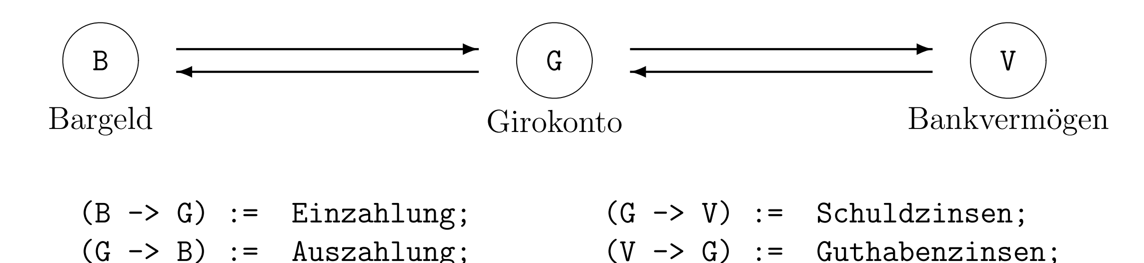
Ein lesender Zugriff auf die Konzentrationen in den Behältnissen ist ebenfalls ohne Eingriff in die Komponenten `K_1` und `K_2` möglich.

In den beiden letzten Beispielen hat der Verzicht auf Autonomie die Modellbildung erleichtert und zu einer übersichtlichen Modellzerlegung geführt. Ein weiteres Beispiel soll zeigen, daß der Verzicht auf Autonomie auch unerwünschte Folgen nach sich ziehen kann.

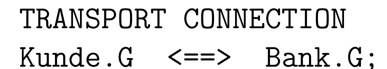
Es soll ein Girokonto mit den dazugehörigen Geldtransfers modelliert werden, das eine Bank ihrem Kunden zur Verfügung stellt.

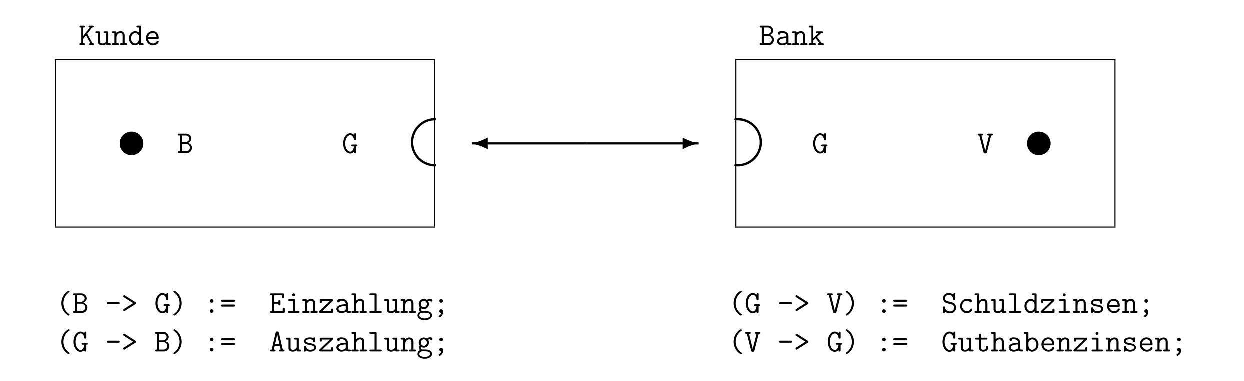
Beispiel: Girokonto

Der Kunde kann Einzahlungen und Abbuchungen vornehmen, die Bank kann Guthabenzinsen und Schuldzinsen verbuchen.




Die Zerlegung soll nach Bank und Kunde vorgenommen werden. Das Girokonto teilen sich die beiden Komponenten.




Da das Girokonto sowohl dem Kunden wie der Bank zugeordnet ist, können beide beliebig darüber verfügen. So kann der Kunde nach Belieben seinen Kreditrahmen überziehen, ohne daß die Bank sich davor schützen kann. Die Bank kann sich nicht einmal davor schützen, daß ein Fremder ihr Vermögen angreift.

Nicht nur zur Herstellung der Eindeutigkeit, auch zum Schutz von Beständen gegen unkontrollierbare fremde Zugriffe ist es sinnvoll einer Komponente bei Bedarf mehr Autonomie zu verleihen.

## 4.6 Transportbezogene Modularisierung mit Teilautonomie

Die bestandsbezogene Beschreibung von Transporten orientierte sich an der klassischen Systemtheorie und an deren Modellbeschreibung durch Definitionsgleichungen. Dadurch war stets die vollständige Autonomie einer Komponente sichergestellt.

Die transportbezogene Beschreibung hingegen erfordert für den Austausch von Elementen zwischen Komponenten einen freien Zugriff auf die Bestände. Die Autonomie für Bestände ist also nicht gegeben. Wir sahen Beispiele, wo dies von Vorteil war, aber auch solche, wo sich dies als nachteilig herausstellte.

Daher erscheint es angebracht, danach zu fragen, wieso sich so unterschiedliche Anforderungen an die Autonomie ergeben.

Autonomie besagt, daß nur eine Komponente selbst Veränderungen an ihren Zustandsvariablen und abhängigen Variablen vornehmen kann. Einflüsse von außen werden einer Komponente nur über ihre Sensorvariablen mitgeteilt. Ob und wie sie darauf reagiert, ist Sache der Komponente, keinesfalls aber ihrer Umgebung.

Dieses Prinzip war eines der Gründe für den Erfolg der Systemtheorie zur Beschreibung von Systemen der Nachrichten- und Regelungstechnik. In den Naturwissenschaften dagegen hat die Systemtheorie bislang kaum Fuß fassen können und wahrscheinlich ist dafür ebenfalls das starre Prinzip der Autonomie verantwortlich. Dafür gibt es folgende Gründe:

In der Technik, aber auch in betriebswirtschaftlichen Organisationsformen, ist man bestrebt, Komponenten so auszulegen, daß sie in ihrem Verhalten von der Umwelt weitgehend unabhängig sind. Technische Komponenten sollen nur auf wenige, durch Schnittstellenbeschreibungen genau festgelegte Eingaben reagieren. Rückwirkungen auf die Eingänge sollten nach Möglichkeit vermieden werden. Durch Komposition von Komponenten kann ein Ingenieur dann größere technische Systeme aufbauen. Wegen der angestrebten Rückwirkungsfreiheit läßt sich das Verhalten des Gesamtsystems sehr einfach aus dem Verhalten der einzelnen Bausteine ermitteln.

In den Naturwissenschaften hingegen findet man zunächst einmal gar keine Komponenten vor. Erst der Forscher grenzt durch sein Bestreben nach Systematik Dinge gegeneinander ab. Diese “Komponenten” sind nun keineswegs autonom. Wenn ein Schläger auf einen Ball trifft, ändert er seine Geschwindigkeit. Der Ball kann sich nicht dagegen wehren. Die Natur fragt nicht, ob für diesen Einfluß von außen eine Schnittstelle vorgesehen war.

In dem einen Fall handelt es sich also um vom Menschen zu seinem eigenen Nutzen geschaffene Systeme, die er frei gestalten kann und durch Prinzipien der Modularisierung und Hierarchisierung versucht überschaubar und verständlich zu halten.

Im anderen Fall handelt es sich um Systeme, die der Mensch vorfindet und zu verstehen und zu erklären versucht. Auf ihre Wirkungsweise hat der Mensch keinen Einfluß.

Diese Überlegungen veranlassen uns, der Autonomie eine besondere Beachtung zu schenken, um die Modellbeschreibungssprache sowohl für den technischen wie für den naturwissenschaftlichen Einsatz attraktiv zu halten.

### 4.6.1 Typisierung der Bestände nach Zugriffsrechten

Die Zugriffsrechte auf einen Bestand entscheiden darüber, inwieweit eine Komponente autonom über einen Bestand verfügen kann. Für Bestände haben wir 4 verschiedene Zugriffsrechte zu unterscheiden:

- Leserecht (L): Recht, die Information auf einem Bestand zu lesen
- Schreibrecht (S): Recht, die Attribute auf einem Bestand zu verändern
- Bringerecht (B): Recht, Elemente einem Bestand zuzuführen
- Holerecht (H): Recht, Elemente von einem Bestand entfernen

Die Ausübung der letzten drei Rechte führt Veränderungen an einem Bestand herbei.

Eine Komponente hat verschiedene Möglichkeiten, mit diesen Rechten umzugehen. Interessant sind vor allem die folgenden Fälle:
Sie kann

- ein Recht selbst wahrnehmen (W) oder
- selbst auf ein Recht verzichten (V)

und sie kann ein Recht

- allen anderen verweigern (K = keiner),
- nur einer einzigen fremden Komponente zugestehen (1) oder
- allen fremden Komponenten zugestehen (A = alle).

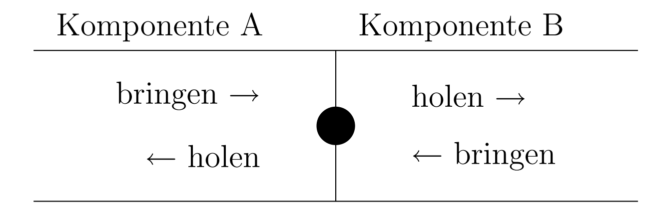
Daraus resultieren sechs Möglichkeiten, jedes einzelne der vier Rechte zu vergeben, insgesamt demnach 24 Typen von Beständen. Die interessantesten davon werden im folgenden diskutiert. Durch Kombination der angegebenen Buchstaben ergibt sich eine Kurzschreibweise für jede der Möglichkeiten.

#### Private Bestände und Durchgangsbestände

Private Bestände und Durchgangsbestände verkörpern zwei Extrempositionen. Private Bestände sind für fremde Komponenten völlig unzugänglich. Auf private Bestände können andere Komponenten keinen Einfluß nehmen. Sie werden mit dem Zusatz PRIVATE versehen. In Anlehnung an Eigenschaftsvariablen erhalten andere Komponenten lediglich ein Leserecht auf private Bestände.
In Kurzschreibweise: (L-WA, SBH-WK).

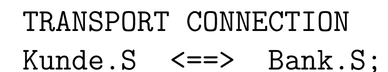
Durchgangsbestände sind für alle Komponenten frei zugänglich und dürfen von allen Komponenten gleichberechtigt mitgenutzt werden. Jede Komponente besitzt alle Rechte. Insbesondere können Transporte in die Komponente und aus der Komponente, d.h. in beiden Richtungen durchgeführt werden. Durchgangsbestände werden daher mit dem Zusatz TRANSIT versehen.
In Kurzschreibweise: (LSBH-WA)

Wenn ein Bestand in zwei oder mehreren Komponenten als Durchgangsbestand deklariert ist, können seine Elemente demnach von allen Komponenten gelesen, geschrieben und entfernt werden und es können Elemente hinzugefügt oder entfernt werden.

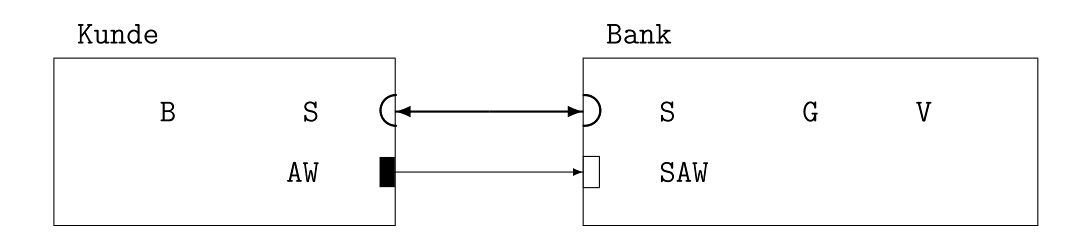



Beispiel: Girokonto

Es wird zusätzlich der Bestand `S` (Schalter) eingeführt und als TRANSIT gekennzeichnet. Die Bank verwaltet das Girokonto ganz allein. Nur auf einen ausdrücklichen Auszahlungswunsch gibt die Bank dem Kunden Geld von seinem Konto, indem sie es auf den Schalter legt. Wünscht er mehr als auf dem Konto ist, verweigert sie die Auszahlung.



```
TRANSPORT CONNECTION
Kunde.S  <==>  Bank.S;
```




```
                  | SAW  falls  SAW <= G
(G -> S) := <
                  | 0    sonst
```

Lediglich der Bestand `S` ist ungeschützt und evtl. auch für einen Dritten zugänglich. Alle übrigen Bestände sind nicht zugänglich.

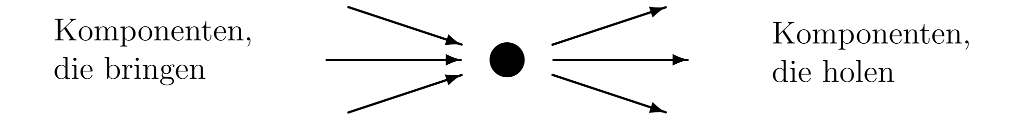
Mit dem Gewinn an vollständiger Autonomie über private Bestände ist auch verbunden, daß sich der eindeutige Zugriff auf die Elemente sicherstellen läßt. Dagegen läßt sich ein eindeutiger Zugriff beim Einsatz von Durchgangsbeständen selbstverständlich nicht gewährleisten.

**Sensorbestände**

Ein wichtiges Anliegen ist es, sich über Bestände in anderen Komponenten zu informieren, ohne diese selbst zu verändern. Zu diesem Zweck muß ein Bestand mit dem Recht zu lesen ausgestattet sein und gleichzeitig auf alle anderen Rechte verzichten. In Anlehnung an Sensorvariablen bezeichnen wir diese Bestände als SENSOR-Bestände. Sensorbestände können die Informationen eines privaten Bestandes und auch aller anderen Typen von Beständen lesen. Der zu lesende Bestand kann auch durch Äquivalenzierung aus mehreren Beständen hervorgehen.
In Kurzschreibweise: (L-WA, SHB-VA)

**Eingangs- und Ausgangsbestände**

Vielfach ist ein Modell so gestaltet, daß Transporte zwischen Komponenten nur in einer Richtung möglich sind. Im allgemeinsten Fall liegt ein richtungsgebundener Transport dann vor, wenn eine oder mehrere Komponenten einen Bestand beschicken und eine oder mehrere Komponenten einen Bestand entleeren.

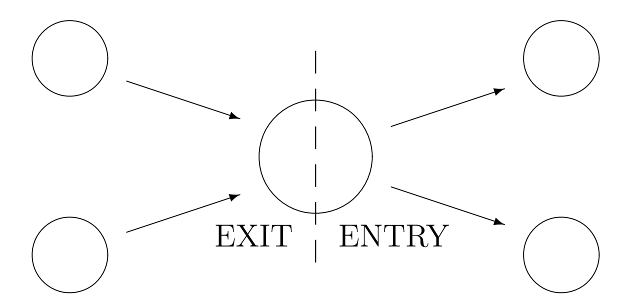



Um zwischen Komponenten, die bringen und Komponenten, die holen unterscheiden zu können, gestalten wir die Rechte derart, daß ein Bestand nur als Ausgangs- bzw. Eingangs-Bestand fungieren kann.

Ein Ausgangsbestand ist ein Bestand, der auf das Recht zu Holen verzichtet und es dafür einem oder mehreren anderen Beständen zugesteht. Gleichzeitig nimmt er das Recht zu Bringen wahr und räumt es ggf. auch weiteren Beständen ein. Um keine Zugriffskonflikte zu provozieren, verzichtet er auf das Recht zu Schreiben, räumt aber fremden Komponenten das Schreibrecht in gleicher Anzahl wie das Holerecht ein.

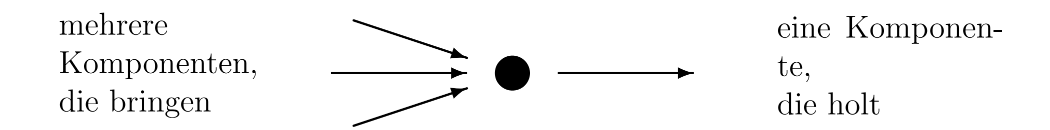
Ein Eingangsbestand ist ein Bestand, der auf das Recht zu Bringen verzichtet und es dafür einem oder mehreren anderen Beständen zugesteht. Gleichzeitig nimmt er das Recht zu Holen wahr und räumt es ggf. auch weiteren Beständen ein. Er verzichtet nicht auf das Schreibrecht und räumt auch anderen Komponenten das Schreibrecht in gleicher Anzahl wie das Holerecht ein.

Mit einer derartigen Verteilung der Rechte kann man sich einen Bestand auch in einen Ausgangs- und einen Eingangsbestand zerlegt vorstellen.

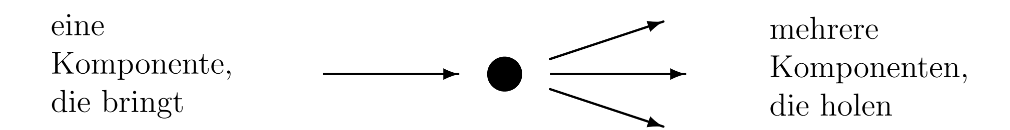


Speziellere Fälle von Transporten liegen vor, wenn der Bestand nur zum Sammeln oder nur zum Verteilen bestimmt ist. Von Übertragung sprechen wir, wenn nur ein einziger Absender und nur ein einziger Adressat zulässig ist.

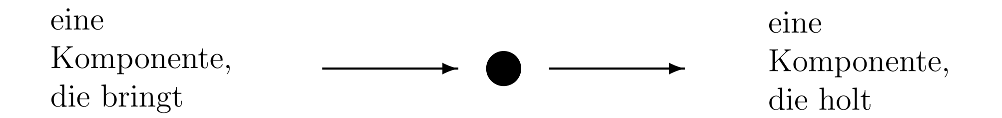
*Sammeln:*




*Verteilen:*




*Übertragen:*




Je nach Verwendungszweck erscheint es sinnvoll, durch gezieltes Einräumen von Rechten, nur die eine oder andere Konfiguration zu erlauben.

Wir unterscheiden daher die folgenden Typen von Ausgangsbeständen:

Ausgangs-Bestand zur Übertragung (EXIT):
Rechte: (L-WA, S-V1, B-WK, H-V1)

Ausgangs-Bestand zum Sammeln (Gathering) (GEXIT):
Rechte: (L-WA, S-V1, B-WA, H-V1)

Ausgangs-Bestand zum Verteilen (Distributing) (DEXIT):
Rechte: (L-WA, S-VA, B-WK, H-VA)

Ausgangs-Bestand zum Sammeln und Verteilen (GDEXIT):
Rechte: (L-WA, S-VA, B-WA, H-VA)

Die Eingangsbestände unterteilen wir in die folgenden Typen:

Eingangs-Bestand zur Übertragung (ENTRY):
Rechte: (L-WA, S-WK, B-V1, H-WK)

Eingangs-Bestand zum Sammeln (Gathering) (GENTRY):
Rechte: (L-WA, S-WK, B-VA, H-WK)

Eingangs-Bestand zum Verteilen (Distributing) (DENTRY):
Rechte: (L-WA, S-WA, B-V1, H-WA)

Eingangs-Bestand zum Sammeln und Verteilen (GDENTRY):
Rechte: (L-WA, S-WA, B-VA, H-WA)

### Erzielen eindeutigen Verhaltens

Das Modellverhalten bei schreibenden oder holenden Zugriffen auf einen Bestand wird immer dann eindeutig beschrieben, wenn nur eine einzige Komponente den schreibenden oder holenden Zugriff ausführt. Folgende Typen von Beständen garantieren daher eine eindeutige Beschreibung: PRIVATE, EXIT, GEXIT, ENTRY, GENTRY.

### Äquivalenzierung von Beständen verschiedenen Typs

Aufgrund der zugeteilten Rechte kann bei Äquivalenzierung in einer höhreren Komponente geprüft werden, ob das Zusammenlegen der Bestände erlaubt ist oder nicht.

Folgende Verbindungen sind erlaubt:

```
PRIVATE  ==  SENSOR

EXIT     ==  SENSOR, (1 x ENTRY  |
                      1 x GENTRY |
                      1 x DENTRY |
                      1 x GDENTRY)

GEXIT    ==  SENSOR, (1 x ENTRY  |
                      1 x DENTRY |
                      1 x GENTRY,  (m1-1) x GEXIT, m2 x GDEXIT |
                      1 x GDENTRY, (m1-1) x GEXIT, m2 x GDEXIT |
                      1 x TRANSIT)

DEXIT    ==  SENSOR, (1  x ENTRY |
                      1  x GENTRY |
                      n1 x DENTRY, n2 x GDENTRY)

GDEXIT   ==  SENSOR, (1  x ENTRY |
                      1  x GENTRY , m1 x GEXIT, (m2-1) x GDEXIT |
                      n1 x DENTRY,  n2 x GDENTRY |
                      n2 x GDENTRY, (m2-1) x GDEXIT, k x TRANSIT)

TRANSIT  ==  SENSOR, (n1 x DENTRY,  n2 x GDENTRY  |
                      n2 x GDENTRY, m2 x GDEXIT, (k-1) x TRANSIT)
```

Da man sowohl auf der Ausgangsseite wie auf der Eingangsseite bestimmen kann, ob Sammeln oder Verzweigen erlaubt sein soll, ergibt sich eine relativ große Zahl von möglichen Kombinationen. Eine so hohe Anzahl ist für den praktischen Einsatz aber nicht mehr überschaubar.

### 4.6.2 Eingrenzung der Typenzahl für Eingangs- und Ausgangsbestände

Will man dem Anwender lediglich die Möglichkeit bieten, eine eindeutige Modellbeschreibung zu erzwingen - etwa weil keine Absprache zwischen parallel liegenden Eingängen vorgesehen ist, dann genügt eine Unterscheidung der Eingangsbestände in GENTRY und GDENTRY. Für die Ausgangsbestände ist der Typ GDEXIT ausreichend, da ein Bringen stets ohne Verlust der Eindeutigkeit erfolgen kann.

Wir wollen in Zukunft auf die "große Lösung" ganz verzichten und riskieren daher keine Mißverständnisse, wenn wir die Schlüsselwörter im Namen ändern. Wir kürzen diese Bezeichnungen bis auf das 'D' für 'distributed entry' und unterscheiden nur noch:

| | | | |
|---|---|---|---|
| Ausgangsbestand: | EXIT | statt | GDEXIT |
| Eingangsbestand: | ENTRY | statt | GENTRY |
| verteilter Eingangsbestand: | DENTRY | statt | GDENTRY |

Damit ergeben sich die folgenden, einfacheren Regeln für eine mögliche Äquivalenzierung:

```
PRIVATE  ==  SENSOR

EXIT     ==  SENSOR, (1  x ENTRY,  (m-1) x EXIT |
                     n2 x DENTRY, (m-1) x EXIT, k x TRANSIT)

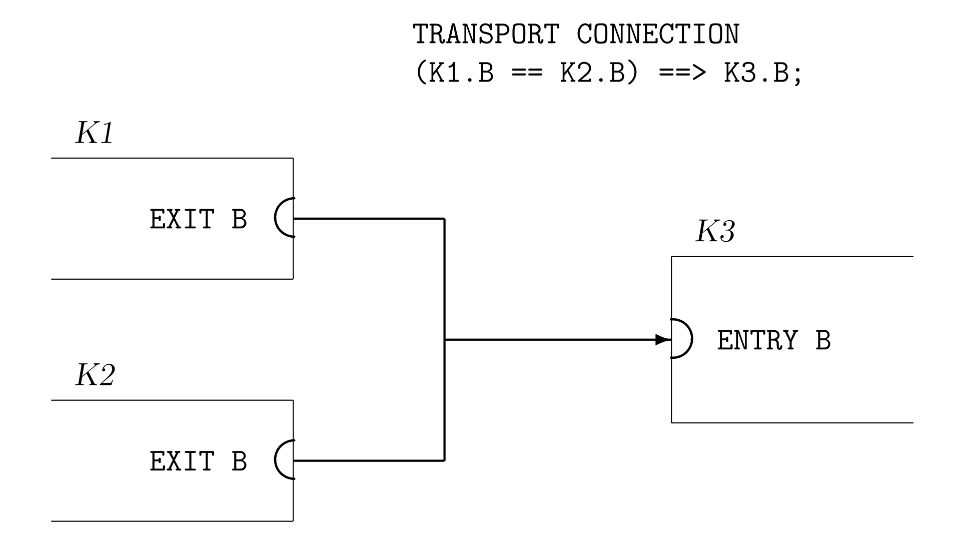
TRANSIT  ==  SENSOR, (n x DENTRY, m x EXIT, (k-1) x TRANSIT)
```

Ein oder mehrere Ausgangsbestände können demnach entweder mit einem (einzigen) Eingangsbestand oder aber mit mehreren Durchgangsbeständen und verteilten Eingangsbeständen verbunden werden.

```
m x EXIT  ==>  ENTRY
m x EXIT  ==>  k x TRANSIT  ==>  n x DENTRY
```

Äquivalenzierungen zwischen Ausgangs- sowie zwischen Eingangsbeständen werden durch den Operator '==' dargestellt, da zwischen diesen Gruppen kein Transport möglich ist. Äquivalenzierungen zwischen Durchgangsbeständen zeigt der Operator '<==>' an. Für alle übrigen Äquivalenzierungen steht der Operator '==>', der in Transportrichtung zu zeigen hat.

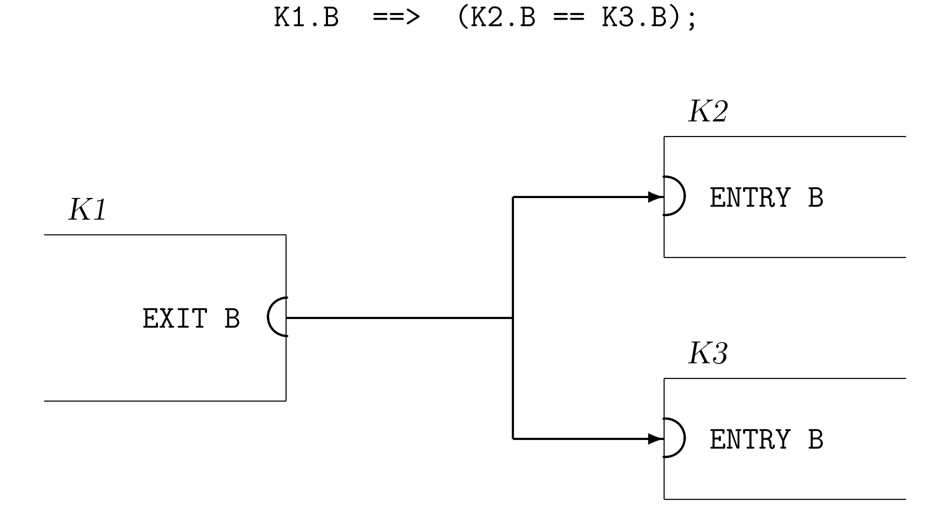
Beispiel: Zwei Ausgangsbestände sind mit einem Eingangsbestand verbunden.




Da nur die Komponente K3 Elemente aus B entnimmt, ist die Eindeutigkeit sichergestellt und kein Zugriffskonflikt möglich.

Das beinhaltet jedoch nicht, daß die Komponente K3 vollständige Autonomie über den Bestand B besitzt. Die Komponente K3 kann schließlich nicht bestimmen, welche Elemente dem Bestand B zugeführt werden.

Beispiel: Ein Ausgangsbestand ist mit zwei Eingangsbeständen verbunden.

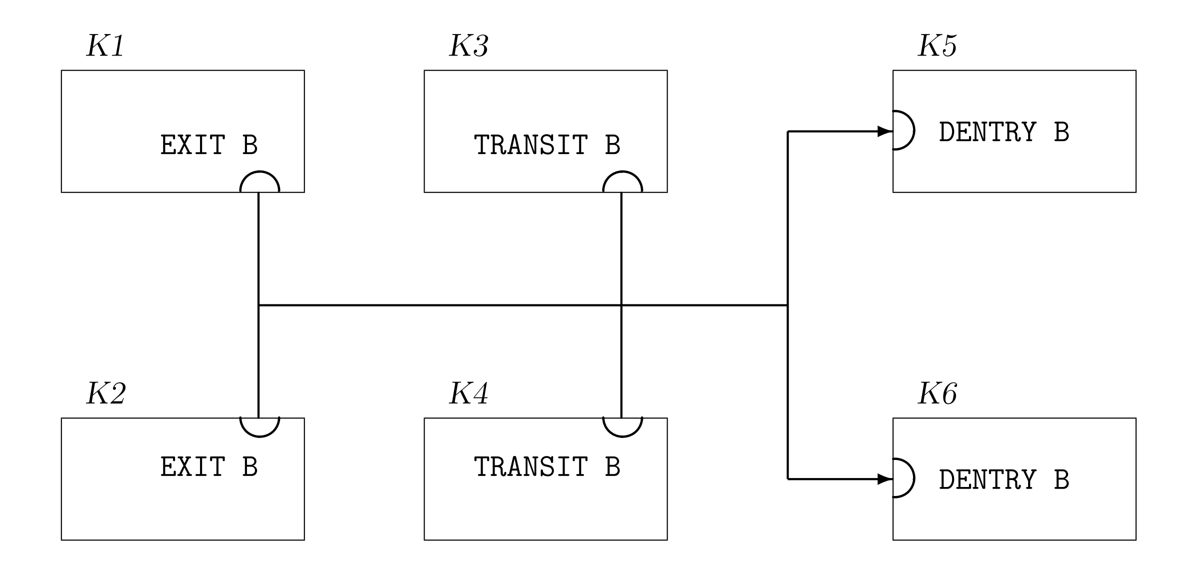



Damit die beiden Komnponenten K2 und K3 nicht gelegentlich zur selben Zeit schreibend oder holend auf dasselbe Element des Bestandes B zugreifen, ist eine Absprache zwischen den beiden Komponenten erforderlich, die über eine zusätzliche Kommunikation erfolgen muß.

Beispiel:

Je zwei EXIT-, TRANSIT- und DENTRY- Bestände sind miteinander verbunden.

```
TRANSPORT CONNECTION
(K1.B == K2.B)  ==>  (K3.B  <==>  K4.B)  ==>  (K5.B == K6.B);
```




Innerhalb der Komponente, in der ein Bestand mit den genannten Attributen deklariert wird, sind die folgenden Operationen zulässig:

| | Antransport | Abtransport | Attribute setzen | Attribute lesen |
|---|---|---|---|---|
| PRIVATE | X | X | X | X |
| EXIT | X | | | X |
| ENTRY | | X | X | X |
| DENTRY | | X | X | X |
| TRANSIT | X | X | X | X |

Die Erfahrung zeigt, daß die zuletzt vorgenommene Unterscheidung in die Bestandstypen EXIT, ENTRY und DENTRY für praktische Anwendungen ausreichend ist und in der Regel nur wenig Bedarf für eine detailliertere Vergabe der Rechte besteht.

Jede der beteiligten Komponenten kann die Transportrichtung festlegen und damit eine Überprüfung der Verbindung veranlassen, ob denn eine weitere Komponente beteiligt ist, die einen Transport in umgekehrter Richtung vornimmt. Darüber hinaus kann eine Komponente, die Elemente hereinholt, verfügen, daß keine weitere Komponente Hole- oder Schreibrecht auf diese Elemente erhält. Auf diese Weise läßt sich ein eindeutiger Zugriff sichern.

Während eine Komponente mit der Vereinbarung eines Bestands vom Typ TRANSIT auf ihre Autonomie vollständig verzichtet, tritt sie mit der der Deklaration eines Eingangs- oder Ausgangsbestandes nur soviel Autonomie wie nötig an fremde Komponenten ab.

Wir meinen damit eine annehmbare Lösung gefunden zu haben, die den unterschiedlichen Fällen der Modellierungspraxis gerecht wird. Der Sprachumfang wird dadurch nur geringfügig erweitert, die Sicherheit beim Modellieren und das Verständnis des Modells jedoch sehr gefördert.

#### Zahlenmäßige Eingangs- und Ausgangsbestände

Wenn Bestände nicht durch individuelle Elemente sondern durch einen Zahlenwert ausgedrückt werden, ist die eben vorgenommene Unterscheidung in verschiedene Bestandstypen ebenfalls interessant.

Zwar spielt der eindeutige Zugriff auf einen Eingangsbestand in diesem Fall keine Rolle, dafür ist es aber von Bedeutung, daß aus einem Eingangsbestand nicht mehr Elemente entnommen werden als darin enthalten sind (siehe Kapitel Anwendungen: Petrinetze und Philosophenproblem). Dies kann der Benutzer leichter sicherstellen, wenn nur eine Komponente aus einem Bestand entnimmt. Auch in diesem Fall müßten sich die Komponenten sonst gegenseitig abstimmen. Wenn dies nicht vorgesehen ist, kann durch die Deklaration eines Bestands mit dem Typ ENTRY ausgeschlossen werden, daß eine eine zweite Komponente den Bestand vermindert.

#### Nicht angeschlossene Eingangs- und Ausgangsbestände

Eine andere Problemstellung ergibt sich aus der Frage, wie verfahren werden soll, wenn ein Eingangs- oder Ausgangsbestand nicht mit einem Bestand in einer anderen Komponente verbunden ist. Sinnvollerweise wird in einem solchen Fall so verfahren als würde eine angeschlossene Komponente keine Aktivität zeigen. Das bedeutet, ein Eingangsbestand erhält seine Vorbesetzung, die auch entnommen werden kann, und ein Ausgangsbestand sammelt all die Elemente, die ihm zugeführt werden, ohne sie zu vernichten.

### 4.6.3 Hierarchiebildung mit Beständen

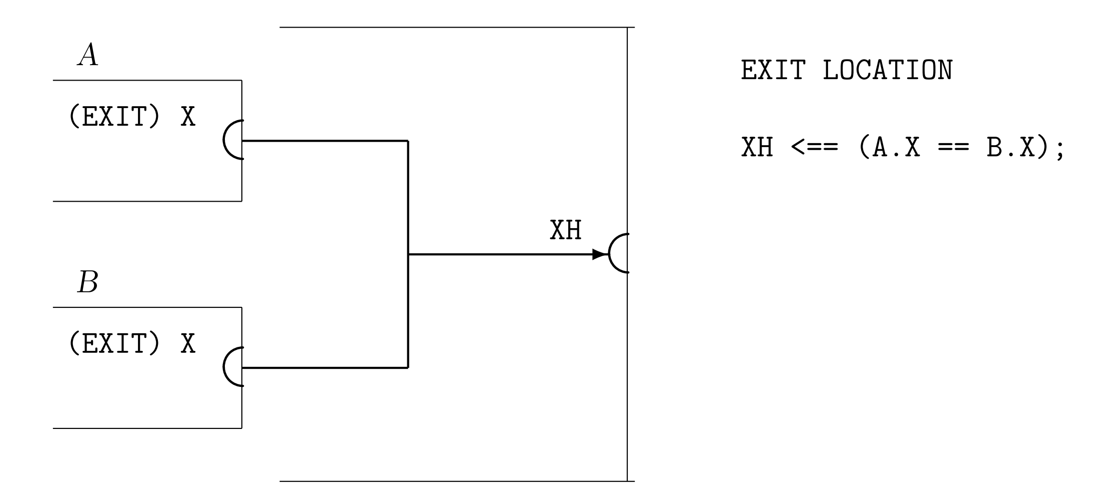
Hierarchiebildung mit Eigenschaftsvariablen haben wir bereits behandelt. Selbstverständlich sollen auch Bestände kein Hindernis darstellen, Hierarchiebildung zu betreiben.

Wir müssen es daher möglich machen, daß auch eine höhere Komponente eine eine Schnittstelle mit ENTRY, DENTRY, TRANSIT oder EXIT LOCATIONS anlegen kann. Aus der Sicht einer übergeordneten Komponente darf es dabei keinen Unterschied machen, ob eine derartige Location einer Basiskomponente oder einer höheren Komponente angehört.

In Anlehnung an Eigenschaftsvariablen verbinden wir die Deklaration gleich mit der/den entsprechenden Verbindungen zu den darunterliegenden Komponenten.

**EXIT-LOCATIONS**

Eine EXIT-Location für eine höhere Komponente kann durch Verbindung mit einer oder mehreren EXIT-Locations aus Subkomponenten vereinbart werden.



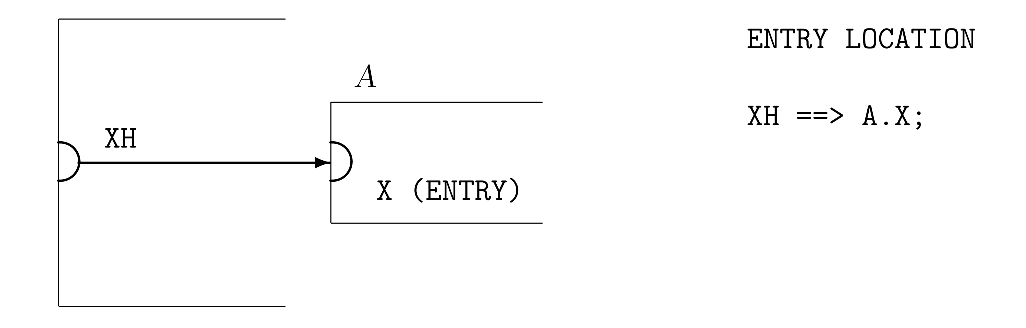

```
EXIT LOCATION

XH <== (A.X == B.X);
```

Da die neu vereinbarte EXIT-Location nach außen hin auf das Recht zu Holen verzichtet, dürfen A.X und B.X mit keiner Location eines anderen Typs (ENTRY, DENTRY oder TRANSIT) verbunden sein.

**ENTRY-LOCATIONS**

Eine ENRTY-Location für eine höhere Komponente kann durch Verbindung mit einer einzigen ENRTY-Location aus einer Subkomponente vereinbart werden.

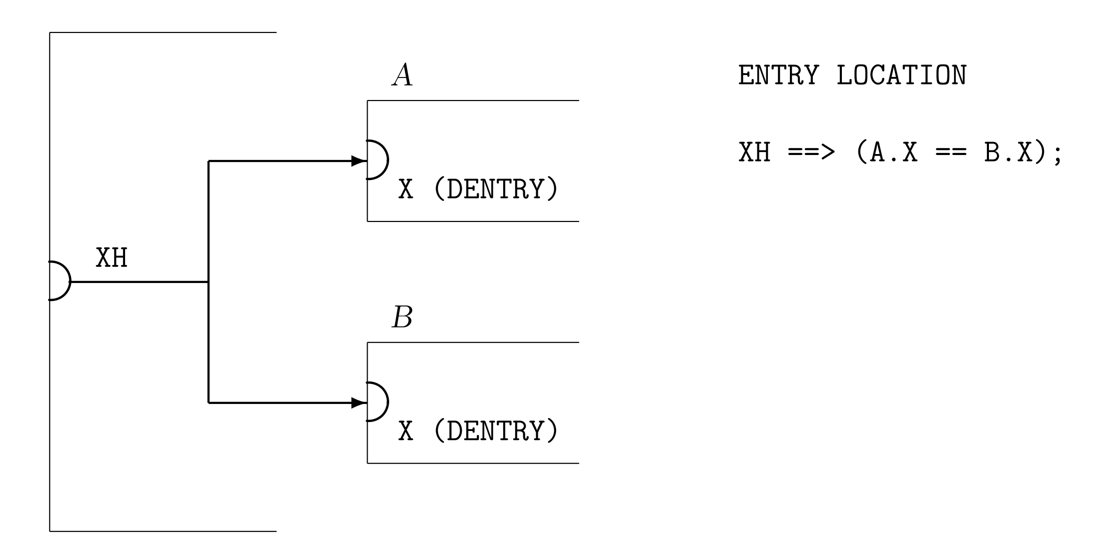



```
ENTRY LOCATION

XH ==> A.X;
```

Alternativ kann eine ENTRY-Location auch durch Verbindung mit einer oder mehreren DENTRY-Locations aus Subkomponenten vereinbart werden.

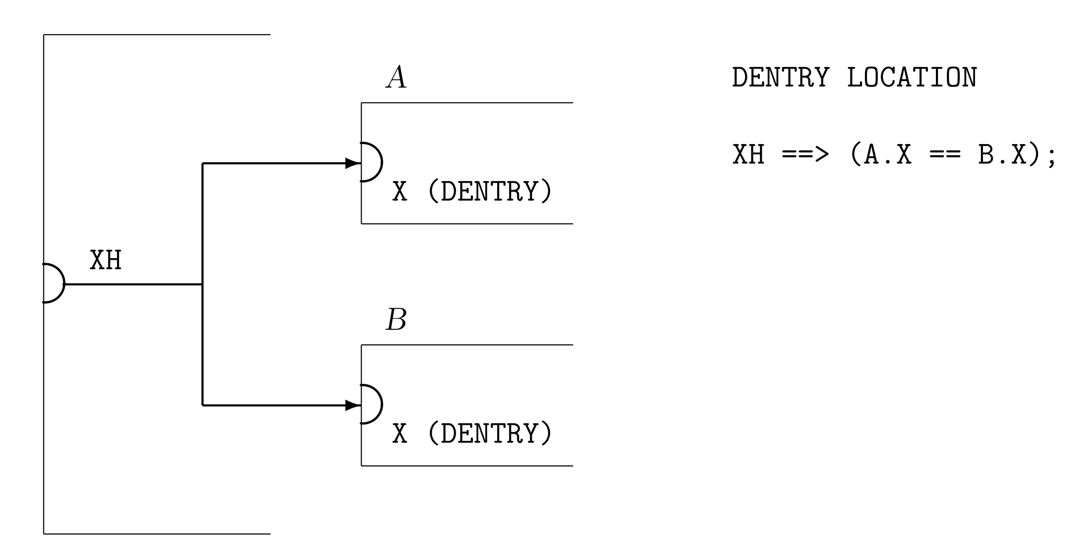
ENTRY-Locations üben das Recht zu holen nach außen hin exklusiv aus. Innerhalb der Komponente dürfen aber mehrere Eingangs-Locations vom Typ DENTRY von diesem Bestand holen. Durch eine geeignete Kommunikation kann innerhalb der höheren Komponente ein eindeutiger Zugriff hergestellt werden.




Da die neu vereinbarte ENTRY-Location nach außen hin auf das Recht zu Bringen verzichtet, darf innerhalb der höheren Komponente keine Verbindung zu Locations anderen Typs (EXIT oder TRANSIT) hergestellt werden.

**DENTRY-LOCATIONS**

Eine DENTRY-Location für eine höhere Komponente kann durch Verbindung mit einer oder mehreren DENTRY-Locations aus Subkomponenten vereinbart werden.




Da die neu vereinbarte DENTRY-Location nach außen hin auf das Recht zu Bringen verzichtet, darf innerhalb der höheren Komponente keine Verbindung zu Locations anderen Typs (EXIT oder TRANSIT) hergestellt werden.

**TRANSIT-LOCATIONS**

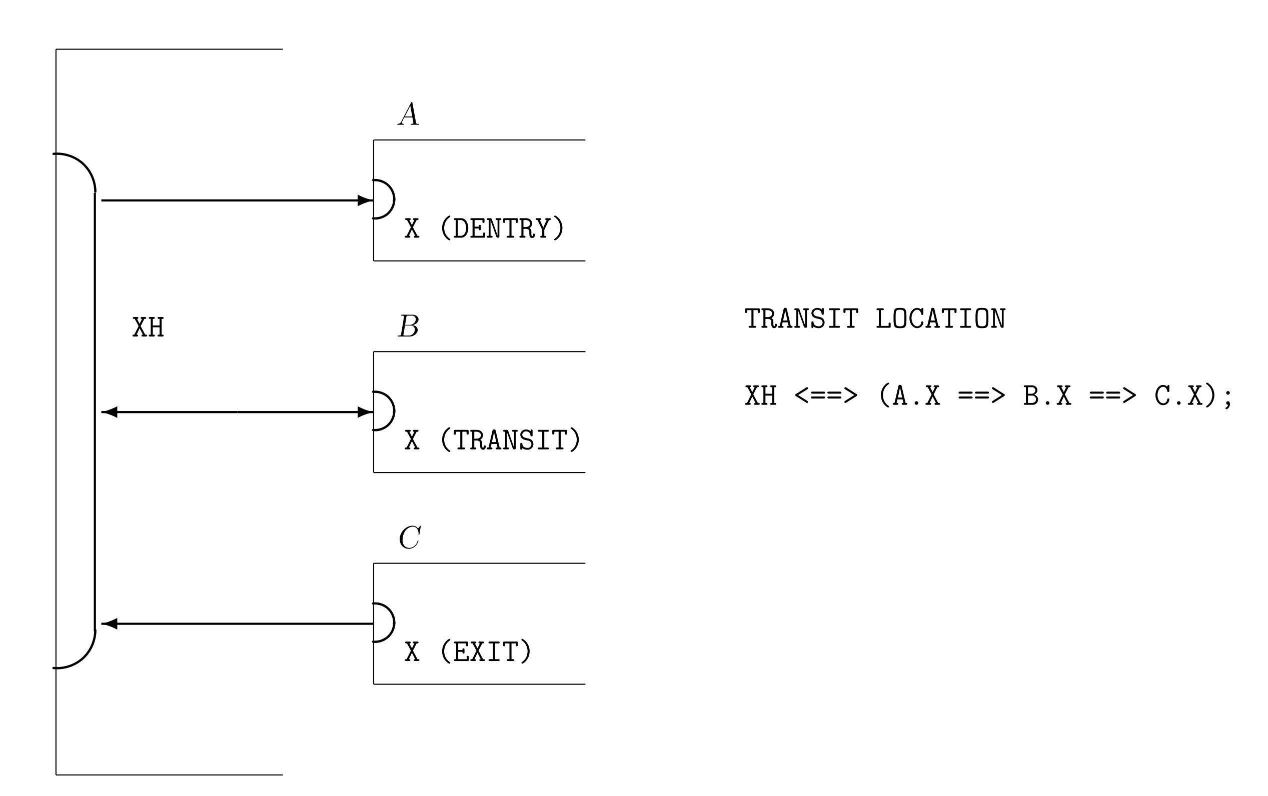
Eine TRANSIT-Location kann in einer höheren Komponente durch Verbindung mit

- einer oder mehreren TRANSIT-Locations oder
- einer Kombination aus EXIT- und DENTRY-Locations

vereinbart werden.
Zusammen mit einer TRANSIT-Location können auch einzelne EXIT- oder DENTRY-Locations mit einbezogen werden.

Anders ausgedrückt: Eine TRANSIT-Location kann durch Verbindung mit einer einzelnen TRANSIT-Location aus einer Subkomponente oder durch Verbindung mit einer Transportverbindung deklariert werden.

Da die TRANSIT-Location nach außen hin sowohl ein Bringerecht wie ein Holerecht gewährt, muß sie innerhalb der höheren Komponente mit Beständen verbunden sein, die ebenfalls diese Rechte gewähren.




Die Notation zerlegt sich in zwei Teile: Der rechte, geklammerte Teil zeigt eine Transportverbindung, d.h. Verbindungen, wie sie auch ohne die eingeführte TRANSIT LOCATION bestehen würden. Der linke Teil stellt die Verbindung zur neuen Location her.

## 4.7 Quellen und Senken

Wir hatten bisher nur geschlossene Modelle betrachtet, d.h. Modelle, die eine bestimmte Anzahl von Elementen in einem geschlossenen Kreislauf enthielten. Sieht man im Modell ein Reservoir für eben nicht benötigte Elemente vor und dimensioniert es genügend groß, kann man auf das Erzeugen und Vernichten von Elementen verzichten.

Die Einführung eines Reservoirs mutet in manchen Fällen jedoch künstlich an, wenn das Reservoir im modellierten System keine Entsprechung besitzt. Vor allem in einem modular aufgebauten Modell müßten u.U. über mehrere Komponenten hinweg Rücktransporte durchgeführt werden. Auch sollten mobile Elemente wieder ihre Vorbesetzung erhalten, wenn sie dem Reservoir entnommen werden.

Es sind daher Vorkehrungen zu treffen, damit neue Elemente erzeugt und bestehende Elemente vernichtet werden können. Bei der bestandsbezogenen Beschreibung, die einen Transport ohnehin in eine Operation zum Erzeugen und eine zum Vernichten zerlegt, ist hierfür lediglich ein Zugang zu neuen Elementen zu schaffen.

Beispiel: Erzeugen: `B_1^ := B_1 + NEW Job {1..5};`
Vernichten: `B_2^ := B_2 - B_2: Job{1};`

Bei Verwendung der transportbezogenen Beschreibung wird die Einführung von Quellen und Senken erforderlich.

Eine Quelle ist ein Bestand auf dem sich potentiell unendlich viele Elemente befinden. Eine Komponente darf ihr beliebige Elemente entnehmen, aber keine hinzufügen.
Die Rechte eines Quellbestandes lauten daher: (L-WK, S-WK, B-VK, H-WK).

Lesen und Schreiben ist gestattet. Das Schreiben ermöglicht eine von der Vorbesetzung abweichende Anfangsbelegung der mobilen Komponenten. Auch als konstant vereinbarte Zustandsvariablen dürfen an dieser Stelle gesetzt werden.

Eine Senke ist ein Bestand, dem in unbegrenzter Menge Elemente hinzugefügt, aber nicht mehr entnommen werden dürfen.
Die Rechte einer Senke lauten daher: (L-VK, S-VK, B-WK, H-VK).
Auch das Lesen und Schreiben ist untersagt. Damit sind Elemente, die auf eine Senke gebracht wurden, quasi nicht mehr existent.

Eine Quelle wird als SOURCE LOCATION, eine Senke als SINK LOCATION deklariert. Vorbesetzungen mit einer bestimmten Menge (von mobilen Komponenten) sind nicht möglich. Weder Quellen noch Senken sind anderen Komponenten zugänglich.

Quellen und Senken dürfen überall dort verwendet werden, wo sonst auch Bestände stehen dürfen. Da eine Quelle potentiell unendlich viele Elemente enthält, darf jedoch nie auf die Quelle als Ganzes zugegriffen werden. Eine Quelle ist stets auf die nötigen Elemente zu beschneiden.

Beispiele:

1)
```
SOURCE  LOCATION  Wasserhahn (REAL [l])
PRIVATE LOCATION  Becher     (REAL [l])
SINK    LOCATION  Ausguss    (REAL [l])

(Wasserhahn -> Becher)' :=  0.1 [l/s];     # kontinuierlicher Fluss
                                           # in einen Becher

(Becher -> Ausguss)'    :=  0.5 [l/s];     # Ausschuetten des Bechers
```

2)
```
SOURCE  LOCATION  Lieferant (INT)
PRIVATE LOCATION  Lager     (INT)
SINK    LOCATION  Kunde     (INT)

(Lieferant -> Lager) :=  Liefermenge;    # diskreter Zugang in ein Lager
                                         # bei Lieferung (Liefermenge > 0)

(Lager -> Kunde)     :=  Abholmenge;     # diskreter Abgang aus einem Lager
                                         # bei Abholung (Abholmenge > 0)
```

3)
```
SOURCE  LOCATION  Auftragsquelle (SET OF Job),
PRIVATE LOCATION  Queue          (SET OF Job),  Server (SET OF Job)
SINK    LOCATION  Auftragssenke  (SET OF Job)

IF  T >= TGenera
LET
    Auftragsquelle: Job{1} -> Queue;    # Entnahme von Auftraegen aus
    TGenera^ :=  T + TAbst;             # einem Quellvorrat
END

IF  CARD Server  <  NUnits
LET
    FOREACH  J  IN  Queue
    LET
        J -> Server;
        J.TReady^ :=  T + TWork;
    END
END

FOREACH  J  IN  Server  |  T >= J.TReady
LET
    Server: Job{1} -> Auftragssenke;     # Ausschleusen eines Auftrags
END                                      # auf eine Senke
```

Quelle und Senke sind eigenständige Typen von Beständen und lassen sich nicht einfach als Spezialfälle von ENTRY- und EXIT-Locations auffassen. Mit Quellen und Senken ist nämlich eine eigene Dynamik verbunden, d.h. eine Dynamik, die nicht vom Benutzer definiert ist.

Die Implementierung sorgt dafür, daß mobile Komponenten, die an einer Senke ankommen, vernichtet werden, um nicht unnötig Speicherplatz zu belegen. Eine mobile Komponente wird implizit erzeugt, wenn auf ein Element einer Quelle zugegriffen wird, das noch nicht vorhanden ist.

## 4.8 Beziehungen zu System Dynamics

Die in diesem Kapitel vorgenommene Unterteilung der Modellvariablen in Eigenschaftsgrößen und Bestandsgrößen wurde in einem anderen Zusammenhang bereits Ende der sechziger Jahre von J.W. Forrester angeregt.

Forrester beschäftigte sich mit Modellen, deren Ergebnisse hohe (firmen-) politische Prisanz hatten: Entwicklung von Industriekomplexen, Stadtentwicklung, Weltmodelle.

Da diese Modelle auf relativ vagen Daten beruhten und auch nicht klar war, ob alle wesentlichen Einflußgrößen Berücksichtigung fanden, hatte Forrester die Notwendigkeit, auf möglichst anschauliche Art und Weise die Zusammenhänge in seinen Modellen offenzulegen.

Er wollte zeigen, daß das prinzipielle Verhalten dieser Modelle eher von deren Struktur als von einer exakten Parametrierung beeinflußt wird. Auch wenn einzelne Entwicklungen schneller oder langsamer verlaufen als angenommen, es werden immer dieselben Folgen eintreten.

Um diese Strukturen (z.B. positive oder negative Rückkopplungen) anschaulich darzustellen, entstand eine graphische Darstellungsmethode, die unter dem Namen *System Dynamics* [Forr 72a, Forr 72b, Forr 73] bekannt wurde. Darüber hinaus wurde auch die Simulationssprache DYNAMO [Crae 85] entwickelt, die eine unmittelbare Umsetzung einer System Dynamics Darstellung in eine Programmiersprache erlaubt, und die auch heute noch Verwendung findet.

Wegen der vagen Daten verzichtet DYNAMO bzw. System Dynamics auf Differentialgleichungen und begnügt sich mit Differenzengleichungen zur Beschreibung des dynamischen Modellverhaltens. Auch Ereignisse sind nicht vorgesehen, weil auch hierzu präzise Angaben notwendig sind. Die Modelle dienten schließlich in erster Linie dazu, mögliche Entwicklungen aufzuzeigen und sollten keine exakten Vorhersagen liefern.

Gerade wegen dieser Beschränkungen hat System Dynamics außerhalb der Wirtschafts- und Sozialwissenschaften aber nie Beachtung gefunden oder gar Bedeutung erlangt. Dies ist schade, da es nur wenige Methoden gibt (etwa Blockdiagramme der Regelungstechnik, Bondgraphen), welche es erlauben, aus der graphischen Modelldarstellung den vollständigen Satz von Modellgleichungen abzuleiten. Um tatsächlich über ein Modell diskutieren zu können, ist dies aber eine Notwendigkeit. Die meisten anderen graphischen Modelldarstellungen geben nur Teilaspekte eines Modells wieder.

Wegen der methodologischen Ähnlichkeiten zu dem hier diskutierten Sprachkonzept soll an dieser Stelle eine kurze Einführung in System Dynamics gegeben werden.

### Einführende Darstellung

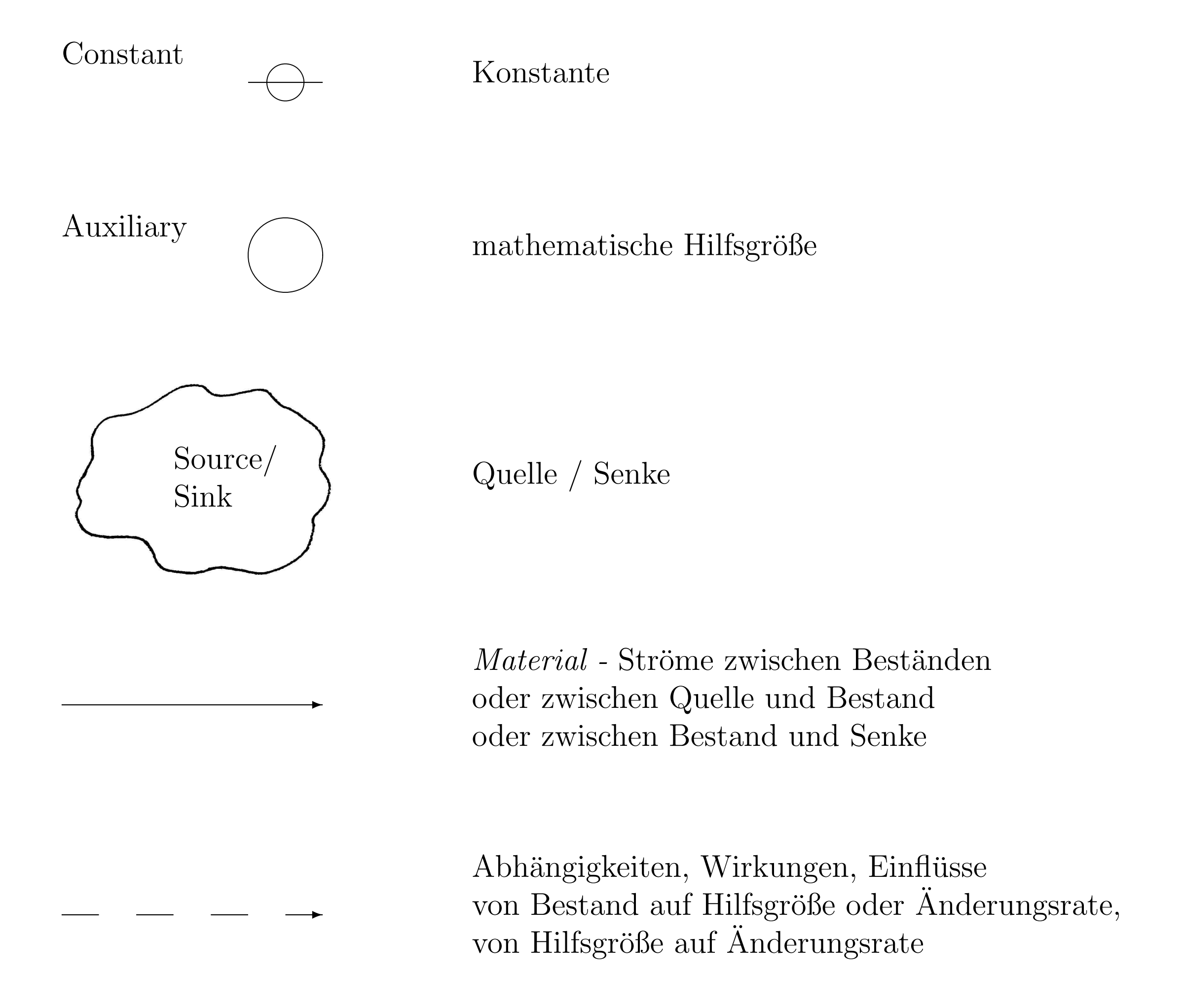
System Dynamics führt die folgenden Symbole ein:

Level

Bestand, Inhaltsgröße

Rate

Veränderungsrate von Beständen
(Stromstärke: Menge im Zeitschritt $\Delta t$)




Die Zustandsgrößen werden stets als Bestände angesehen. Die Bestände wirken auf Hilfsgrößen ein. Aus Hilfsgrößen und Beständen können weitere Hilfsgrößen abgeleitet werden. Schließlich nehmen die Bestände und Hilfsgrößen Einfluß auf die Änderungsraten der Bestände. Die Wirkungen der Variablen aufeinander werden durch gestrichelte Pfeile dargestellt.

Die Veränderung eines Bestandes hat einen Strom zur Folge, der in einem anderen Bestand oder einer Senke mündet und in einem anderen Bestand oder einer Quelle seinen Ursprung hat. Die Ströme werden als durchgezogene Linien dargestellt.

Ein kleines Beispiel soll den Einsatz der Methode verdeutlichen.

Beispiel: Modell *Lagerhaltung*

Die Abb. 4.8-1 zeigt den Warenfluß in einem einfachen Lagerhaltungsmodell: Waren werden bei einem Unternehmen bestellt (Bestellrate) und aus dem Bestellbestand treffen mit einer gewissen zeitlichen Verzögerung gelieferte Waren im Lager ein (Lieferrate). Durch Einkäufe der Kunden gehen schließlich Waren aus dem Lager ab (Abgangsrate).

Obwohl Zugänge und Abgänge wohl allgemein als diskret angesehen werden, werden sie hier durch (stetige) Ströme beschrieben. Es kommt eben weniger darauf an, einzelne Waren zu erfassen, als vielmehr den gesamten Waren- und den daraus resultierenden Kapitalstrom zu betrachten.

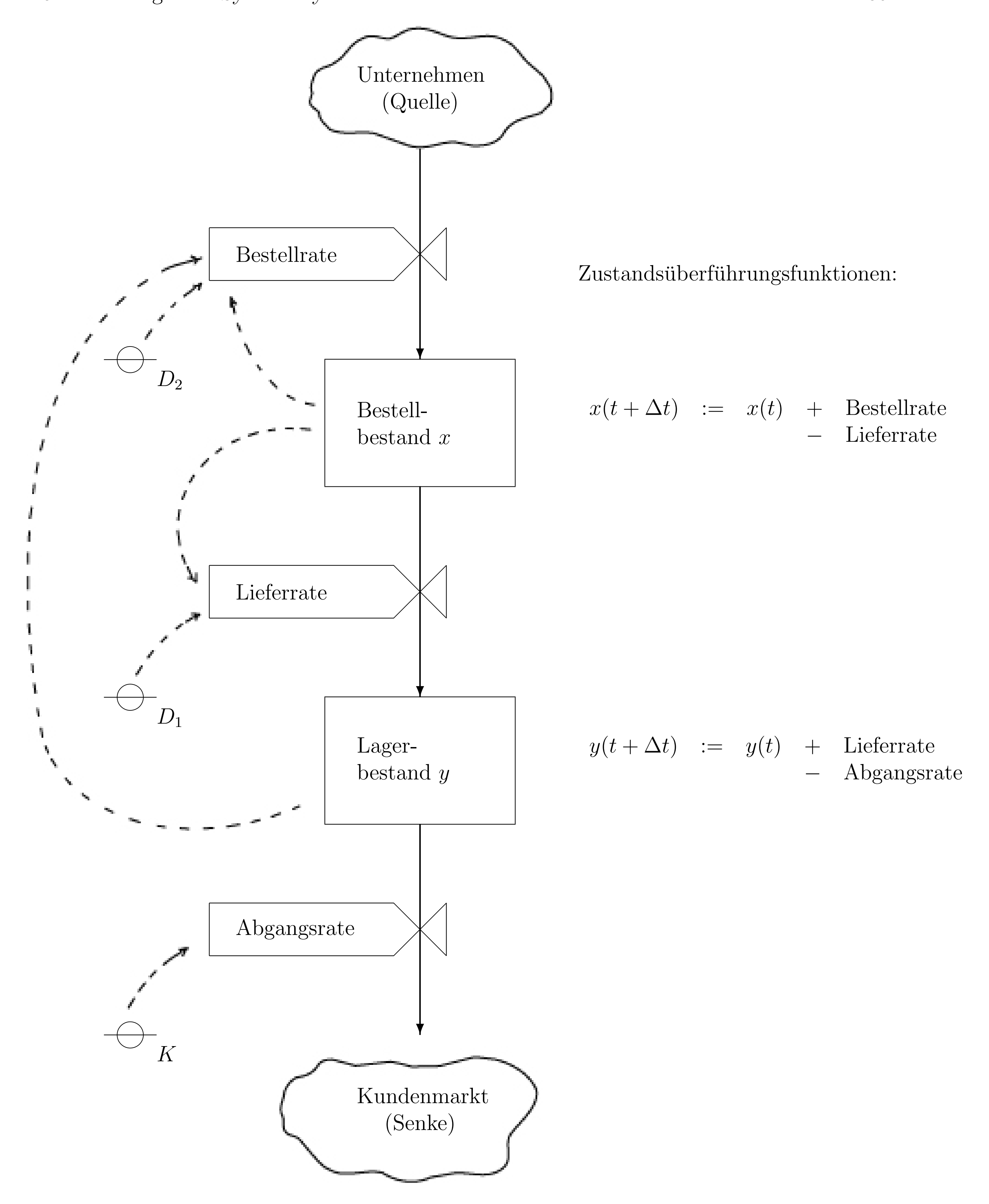



Abb. 4.8-1: Modell *Lagerhaltung* in System Dynamics Darstellung

## Verwandtschaft zum diskutierten Sprachkonzept

Wie gerade im gezeigten Beispiel deutlich wird, stellt die durch ein Ventil symbolisierte Änderungsrate einen Spezialfall unserer Transportfunktion dar.

Während wir die Transportfunktion für kontinuierliche, zeitdiskrete und elementweise Transporte eingeführt haben, ermöglicht der mit Differenzengleichungen beschriebene Transport lediglich zeitlich getaktete Übergänge.

Wie wir meinen, ist diese Darstellungsart aber keineswegs an Differenzengleichungen gebunden, da man statt der Änderungsrate auch jede andere Transportfunktion einsetzen kann.

Die Hilfsgrößen entsprechen den abhängigen Größen. Die gestrichelten Pfeile, die auf die Hilfsgrößen und Änderungsraten weisen, sind Wirkpfeile und daher entspricht dieser Teil der graphischen Darstellung unserem Abhängigkeitsgraphen.

Hinzu kommen die durchgezogenen Pfeile, welche Ströme zwischen Beständen repräsentieren, und stets von einer Änderungsrate in ihrer Stärke gesteuert werden.

Auf diese Art werden sowohl Transportvorgänge als auch Wirkungen sehr anschaulich in das gleiche Bild eingebunden. Aus diesem Grund ist System Dynamics (evtl. mit gewissen Erweiterungen) als graphisches Darstellungsmittel für die hier besprochene Modellbeschreibungssprache von hohem Interesse.

Es fällt allerdings auf, daß System Dynamics keine Zustandsgrößen kennt, die Eigenschaften verkörpern. Dies ist natürlich nur statthaft, wenn sich die Eigenschaftsgrößen stets aus den Bestandsgrößen ableiten lassen.

In den wirtschafts- und sozialwissenschaftlichen Gebieten, in denen System Dynamics in erster Linie zum Einsatz kommt, gibt es hiermit offenbar keine Probleme. In anderen Gebieten empfanden wir es jedoch als durchaus sinnvoll, einen Zustand auch durch eine Eigenschaft, d.h. durch einen Wert aus einer Wertemenge ausdrücken zu können.

Diese Frage müssen wir leider offenlassen, da sie noch nicht endgültig ausdiskutiert ist. Ein anderer Punkt, die Modularisierung, läßt sich jedoch nach den Beiträgen der letzten Abschnitte bereits zufriedenstellend abhandeln.

## Modularisierung von System Dynamics Darstellungen

DYNAMO bzw. System Dynamics beinhaltet kein Konzept für eine Modularisierung eines Modells. Modelle sind daher immer nur als Ganzes darstellbar. Dieses Manko fiel bislang nicht allzu sehr ins Gewicht, da auch die anderen Simulationssprachen keine Modellzerlegung ermöglichten.

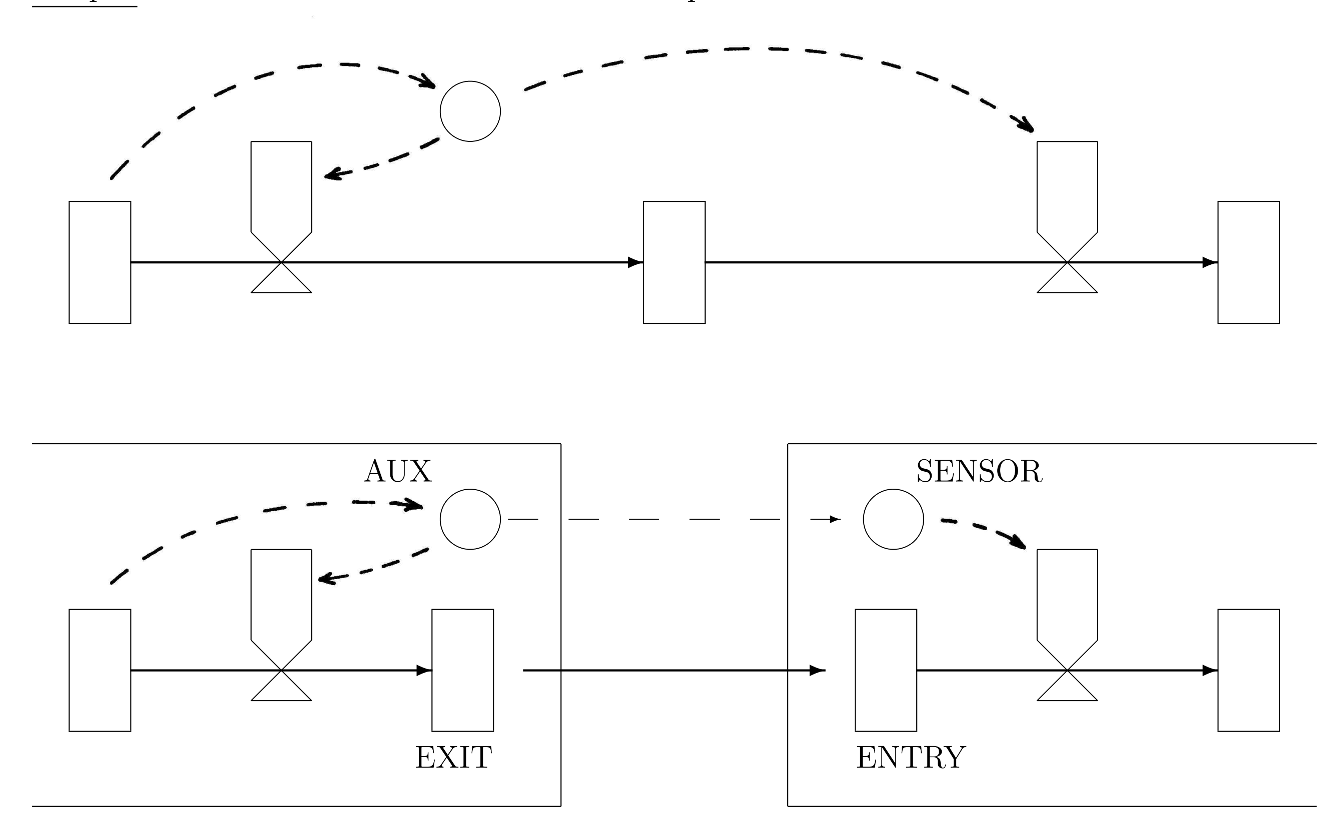
Wie wir in den vorangegengenen Abschnitten gelernt haben, muß die Modularisierung einer transportbezogenen Beschreibung durch Aufteilung eines Bestandes auf zwei oder mehrere Bestände vorgenommen werden.

Um dies auch graphisch zum Ausdruck zu bringen, ist ein Bestand, der mit einem anderen geteilt werden soll, an den inneren Rand einer Komponente zu zeichnen und als ENTRY, DENTRY, EXIT oder TRANSIT zu kennzeichnen.

Auf einer übergeordneten Ebene können dann diese Bestände durch Flußpfeile miteinander verbunden werden. Auch ein hierarchischer Modellaufbau ist denkbar.

Durchtrennt die Komponentengrenze ein Wirkpfeil, dann ist eine Sensorvariable einzuführen und auf nächst höherer Ebene eine Verbindung herzustellen.

Beispiel: Gesamtmodell verteilt auf zwei Komponenten




Wegen der vielen Verwandtschaften zu dem in dieser Arbeit diskutierten Sprachkonzept erweist sich System Dynamics daher als interessanter Ansatzpunkt, um Zusammenhänge und Flüsse im Modell graphisch zu repräsentieren. Um den Anspruch auf Vollständigkeit der Darstellung aufrecht erhalten zu können, wird es wohl noch erforderlich sein, auch Zustandsvariable mit einzuarbeiten, welche Eigenschaften repräsentieren.

Insbesondere ist diese Darstellungsart dazu geeignet, Zusammenhänge im Modell sowohl für Basiskomponenten als auch für höhere Komponenten und hierarchische Modelle durchgängig zu repräsentieren.

## 4.9 Zusammenfassung

Neben den bereits aus Kapitel 2 bekannten Eigenschaftsgrößen wurden die Bestandsgrößen als gänzlich neuer Variablentyp eingeführt. Bestände repräsentieren Mengen von Elementen. Alle Bestände eines Modells sind disjunkt, d.h. jedes Element ist eindeutig einem Bestand zugeordnet.

Transporte zwischen Beständen lassen sich bestandsbezogen oder transportbezogen beschreiben. Die verschiedenen Beschreibungsformen für kontinuierlichen, diskreten sowie mengenweisen Transport wurden diskutiert.

Ein eigenes Problem beim Transport ist der eindeutige Zugriff auf die Elemente bei Verteilung. Um diesen herzustellen, wurden die für die bestandsbezogene bzw. transportbezogene Beschreibung erforderlichen Maßnahmen diskutiert.

Wie die Ausführungen in Kapitel 3 gezeigt haben, ermöglichen Bestände eine besonders geeignete Art der Modularisierung. Auch hier wurde wieder die bestandsbezogene und die transportbezogene Beschreibung miteinander verglichen.

War es bisher nicht möglich der einen oder der anderen Beschreibungsform den Vorzug zu geben, so erweist sich die transportbezogene Beschreibung bei Modularisierung der bestandsbezogenen deutlich überlegen. Aus diesem Grund wird die transportbezogene Beschreibung weiter verfeinert.

Durch eine gezielte Einschränkung der Autonomie einer Komponente konnte erreicht werden, daß

- die Bestände einer Komponente gegen Zugriffe von außen weitgehend geschützt sind und
- auf höherer Ebene anhand der Verbindungen erkannt werden kann, in welche Richtung ein Transport erfolgen kann.

Ein hierarchischer Modellaufbau erwies sich ebenfalls als möglich.

Um auch offene Modelle formulieren zu können, war es schließlich notwendig, Quellen und Senken als eigene Bestandstypen einzuführen.

Zum Schluß wurde gezeigt, daß Modelldarstellungen mit System Dynamics der hier eingeführten Modellwelt sehr nahestehen und daher ein guter Ausgangspunkt für weitere Betrachtungen sind, um eine formal-sprachliche Modellbeschreibung vollständig auch graphisch beschreiben zu können. Wegen methodologischer Unsicherheiten wurde dieses an sich sehr wichtige Thema jedoch nur andeutungsweise behandelt.

# Kapitel 5

# Systematische Darstellung der Modellbeschreibungssprache

In den beiden letzten Kapiteln wurden eine Reihe von möglichen Spracherweiterungen besprochen, um Warteschlangen- und Transportmodelle mit Hilfe des zustandsorientierten Ansatzes beschreiben zu können.

Dabei sollten die verschiedenen Gesichtspunkte deutlich werden, die für einen Sprachentwurf bedeutsam sind. Es wurden unterschiedliche Konzepte und Konstrukte diskutiert und miteinander verglichen. Als Ergebnis dieser Betrachtungen wird nun in diesem Kapitel eine vollständige Modellbeschreibungssprache vorgeschlagen.

Wegen des großen Umfangs und um die wesentlichen Aspekte nicht in Details verschwinden zu lassen, wird die Syntax und Semantik nicht in allen Einzelheiten beschrieben. Die Teile der Sprachsyntax, die keinen Bezug zu den bisherigen Ausführungen haben, werden nur angedeutet. Eine kurze Abhandlung über diese Teile der Sprache findet sich in Kapitel 6. Die Semantik wird angegeben, soweit sie sich nicht von selbst erklärt.

Das Sprachkonzept ist so gestaltet, daß

- der Sprachumfang übersichtlich bleibt
- die universelle Einsetzbarkeit gewährleistet ist
- keine komplizierte (oder gar widersprüchliche) Syntax entsteht
- das generierte Simulationsprogramm nicht ineffizient wird.

Bei der Festlegung des Sprachumfangs waren folgende Überlegungen maßgebend: Kern jeder Modellbeschreibung sind *Definitionsgleichungen* und *Transportanweisungen*.

Sowohl Definitionsgleichungen wie auch Transportanweisungen, bei denen es nur auf die Anzahl von Elementen ankommt, benötigen einen Wert, der sich aus dem aktuellen Modellzustand bestimmen läßt.

Ein solcher Wert läßt sich

- in geschlossener Form
- in tabellarischer Form oder
- durch einen Algorithmus (Funktion bzw. Prozedur)

errechnen.

Für Transportanweisungen mit unterscheidbaren Elementen werden ein oder mehrere mobile Elemente benötigt, die aus einem oder mehreren Beständen auszuwählen sind.

Zur Auswahl einer Menge von mobilen Komponenten kommt

- das Beschneiden einer (geordneten) Location
- das Beschneiden einer (geordneten) Eigenschaftsmenge
  (Bestimmung des Ordnungswerts durch eine Funktion)
- ein Algorithmus (Funktion)

in Betracht. Das Beschneiden der Mengen ist optional.

Aus diesen Überlegungen geht hervor, daß die universelle Einsetzbarkeit nur hergestellt werden kann, wenn auch eine algorithmische (prozedurale) Bestimmung von Werten und von mobilen Elementen (bzw. deren Indices) möglich ist (siehe Kapitel 6). Da man also auf die Integration von Funktionen und Prozeduren in eine allgemeine Modellbeschreibungssprache ohnehin nicht verzichten kann, ist es aber andererseits möglich, den deklarativen Sprachteil unter Hinweis auf (algorithmische) Funktionen und Prozeduren in engen Grenzen zu halten.

So wäre es möglich, auf tabellarische Funktionen oder die Bildung von Eigenschaftsmengen zu verzichten. Ob dies ratsam ist, müssen weitere Erwägungen klären.

Mit der folgenden Darstellung soll dem Leser ein geschlossenes sprachliches Konzept vor Augen geführt werden. Die Sprachsyntax wird in einer solchen Weise angegeben, daß sie möglichst viel von der Sprachsemantik vermittelt. Sie ist daher etwas ausschweifend und in dieser Form nicht für einen Parser zu gebrauchen. Sie läßt sich allerdings jederzeit in eine LR-1 - Grammatik umformen.

## 5.1 Grundlegende Konzepte der Sprache

In einem formalen Modell wird durch eine mathematische Notation beschrieben, welche Zusammenhänge zwischen Beobachtungen an der Realität vorherrschen. Diese Zusammenhänge gelten für alle Beobachtungszeitpunkte.

Eine Beobachtung bezieht sich stets auf ein bestimmtes Gebiet im Anschauungsraum. Idealisiert kann ein Raumgebiet auch ein Punkt, eine Linie oder eine Fläche sein. Innerhalb eines Raumgebiet können wir Eigenschaften oder Bestände (Inhalte) beobachten.

Jede Beobachtung wird durch den Zeitverlauf einer Variablen dargestellt. Demnach unterscheiden wir

- Eigenschaftsvariablen (Attribute, Qualitäten) oder
- Bestandsvariablen (Mengen, Quantitäten).

Einer Eigenschaftsvariablen ist zu jedem Zeitpunkt genau ein Wert aus einer der Wertemenge zugeordnet. Dabei kann es sich um numerische Wertemengen (INT, REAL) handeln, die evtl. durch eine physikalische Maßeinheit ergänzt werden, aber auch logische Werte oder Werte aus benutzerdefinierten Aufzählungsmengen kommen in Betracht.

Einer Bestandsvariablen ist bei gleichartigen Elementen eine Anzahl und bei unterscheidbaren (mobilen) Elementen eine Menge zugeordnet. Bestandsvariablen repräsentieren Inhalte nicht überlappender Raumgebiete. Daher sind ihre Mengen disjunkt, d.h. jedes mobile Element ist genau einem Bestand zugeordnet. Mobile Elemente werden – wegen zahlreicher Verwandtschaften – den Komponenten zugerechnet.

Eine Komponente faßt Modellvariablen und Beziehungen zwischen Modellvariablen so zusammen, daß sie ein vollständiges Modell bilden.

Komponenten unterscheiden wir in die Typen

- Mobile Komponenten
- Basiskomponenten und
- Höhere Komponenten.

```
model_component ::=   mobile_component
                    | basic_component
                    | high_level_component
```

Jede vereinbarte Komponente bildet eine Klasse und kann unter einem eigenen Namen bzw. eigenem Index mehrfach in einem Modell zum Einsatz kommen. Die Ausprägungen einer Klasse besitzen zwar prinzipiell das gleiche dynamische Verhalten, die aktuelle Belegung der Modellvariablen ist jedoch in der Regel verschieden.

Eine mobile Komponente wird als Menge (Verbund) von Variablen vereinbart. Sie darf demnach auch Bestandsvariablen enthalten, auf denen wiederum andere mobile Elemente plaziert sind. Mobile Komponenten repräsentieren lediglich einen Zustand. Sie enthalten keine Beschreibung über dynamische Veränderungen.

Eine Basiskomponente wird ebenfalls als Menge (Verbund) von Variablen vereinbart. Daneben enthält sie auch die Beschreibung für das dynamische Verhalten ihrer eigenen Variablen und für das dynamische Verhalten von mobilen Komponenten auf Bestandsvariablen. Über Sensorvariablen kann sich die Basiskomponente Werte von Variablen fremder Komponenten beschaffen und darauf reagieren. Anders betrachtet, können fremde Komponenten über Sensorvariablen auf das dynamische Verhalten einer Basiskomponente einwirken. Über zugängliche Bestände kann sich die Basiskomponente mobile Komponenten aus fremden Komponenten holen bzw. dorthin bringen.

Zugängliche Bestände können mit Beständen fremder Komponenten zusammengelegt werden. Dadurch ist es möglich, mobile Komponenten in eine fremde Komponente zu bringen bzw. aus einer fremden Komponente zu holen.

Eine höhere Komponente wird als Menge von Basiskomponenten und/oder anderen höheren Komponenten vereinbart. Einflüsse, welche die Subkomponenten gegenseitig aufeinander ausüben, d.h. das Lesen von internen Variablen durch Sensorvariablen, werden durch Wirkpfeile festgelegt. Transporte zwischen den Subkomponenten sind über gemeinsame Bestände möglich, die durch Transportverbindungen hergestellt werden. Zur Einbettung in eine andere höhere Komponente kann eine Schnittstelle geschaffen werden. Variablen der Subkomponenten werden zu diesem Zweck zu eigenen Variablen gemacht.

Auf diese Weise läßt sich eine hierarchische Modellstruktur beschreiben. Auf der untersten Ebene dieser Hierarchie stehen diejenigen Komponenten, die selbst keine weiteren Komponenten enthalten. Das können Basiskomponenten oder mobile Komponenten sein.

Keine Komponente darf in sich selbst oder in einer ihrer Subkomponenten enthalten sein. Die Hierarchie muß zyklenfrei sein. Ein vollständiges Modell muß mindestens eine Basiskomponente enthalten.

Auf den Ebenen der höheren Komponenten und der Basiskomponenten ist die Modellstruktur statisch, auf den Ebenen der mobilen Komponenten ist sie dynamisch. Aus der Sicht der

Informatik bieten mobile Komponenten die gleichen Möglichkeiten wie mehrfach verkettete Listen. Sie stellen daher einen relativ mächtigen und flexiblen Datentyp dar.

## 5.2 Deklaration in Basiskomponenten und mobilen Komponenten

Basiskomponenten und mobile Komponenten besitzen den gleichen Aufbau mit dem Unterschied, daß mobile Komponenten keine Beschreibung der Modelldynamik beinhalten.

```
basic_component ::=    BASIC COMPONENT  identifier
                     [ mobile_subclass_declaration ]
                     [ definition_part ]
                       declaration_part
                     [ dynamic_behaviour ]
                       END OF  identifier

mobile_component ::=   MOBILE COMPONENT  identifier
                     [ mobile_subclass_declaration ]
                     [ definition_part ]
                       declaration_part
                       END OF  identifier
```

Von allen Bestandteilen einer Basiskomponente oder mobilen Komponente ist nur der Deklarationsteil nicht optional. Im ersten Abschnitt werden mobile Komponenten (genauer: Klassen von mobilen Komponenten) deklariert, die in der Komponente benötigt werden.

```
mobile_subclass_declaration ::=   MOBILE SUBCOMPONENT[S] OF CLASS
                                  identifier { ',' identifier }
```

In einem Definitionsteil kann sich der Benutzer eigene Beschreibungsmittel wie

– Dimensionierungskonstanten
– Wertemengen durch Aufzählung
– Maßeinheiten
– tabellarische Funktionen und Verteilungen
– algorithmische Funktionen und Prozeduren
– Schnittstellen zu Funktionen und Prozeduren in der Basissprache C

vereinbaren.

```
definition_part ::=   DEFINITION[S]

                    { DIMENSION  identifier ':=' unsigned_number ';' }

                    { VALUE SET  identifier ':' '(' enum_value_list ')' }
```

```
{ UNIT  identifier ':=' unit_description ';' }

{ TABULAR FUNCTION  tabular_function_definition }

{ TABULAR DISTRIBUTION  tabular_function_definition }

{ FUNCTION  algorithmic_function_definition }

{ PROCEDURE  algorithmic_procedure_definition }

{ C_FUNCTION  c_function_declaration }

{ C_PROCEDURE  c_procedure_declaration }
```

Die Möglichkeiten, die hier geboten werden, sind sehr umfangreich. Sie sind aber notwendig, um den nötigen Komfort und die universelle Einsetzbarkeit zu erzielen. Mehr darüber findet sich in Kapitel 6.

Im Deklarationsteil werden die Modellvariablen, d.h. die Attribute und Bestände der Modellkomponente vereinbart.

```
declaration_part ::=   DECLARATION[S]
                       initial_time_and_unit
                       attribute_declaration_part
                       location_declaration_part
```

Die Zeit wird zwar durch die Standardvariable $T$ repräsentiert und besitzt die Wertemenge REAL, die physikalische Maßeinheit (soweit benötigt) sowie der Anfangszeitpunkt, zu dem auch die angegebene Anfangsbelegung gehört, sind jedoch noch offen.

```
initial_time_and_unit ::=   INITIAL TIME ':='  number  [ unit ] ';'
```

Die Deklaration der Attribute wurde bereits in Abschnitt 2.4.2 behandelt und soll deshalb hier nur noch in aller Kürze angeführt werden.

Attribute erhalten einen eindeutigen Bezeichner und werden durch die Angabe von Eigenschaften näher klassifiziert.

- Verwendung:
  Zustandsvariablen, abhängige Variablen, Sensorvariablen und Zufallsvariablen
- Form der Zeitverlaufs:
  konstant, diskret (sprunghaft) oder kontinuierlich (stetig)
- Wertemenge:
  ganzzahlig, reell, logisch, Aufzählungsmenge, Menge von mobilen Elementen
- physikalische Maßeinheit

```
attribute_declaration_part ::=

            [ STATE     VARIABLE[S]  var_course_specific_declaration ]
            [ DEPENDENT VARIABLE[S]  var_course_specific_declaration ]
            [ SENSOR    VARIABLE[S]  var_course_specific_declaration ]
            [ RANDOM    VARIABLE[S]  var_course_specific_declaration ]

var_course_specific_declaration ::=

            [ CONSTANT   variable_declaration_list ]
            [ DISCRETE   variable_declaration_list ]
            [ CONTINUOUS variable_declaration_list ]
```

Die weitere Syntax ist für Attribute und Bestände identisch und findet sich auf der übernächsten Seite.

Eine Basiskomponente besitzt die volle Autonomie über ihre Eigenschaftsvariablen, d.h. sie allein besitzt das Recht, Veränderungen an den internen Variablen (Zustandsvariablen und abhängige Variablen) vorzunehmen. Andere Komponenten dürfen lediglich lesend auf die internen Variablen zugreifen. Dies ist durch eine entsprechende Vergabe des Lese- und Schreibrechts für jeden Variablentyp geregelt.

| | | |
|---|---|---|
| STATE | L–WA | S–WK |
| DEPENDENT | L–WA | S–WK |
| SENSOR | L–WA | S–VA |
| RANDOM | L–WK | S–VK |

Bestände gelten ebenfalls als Zustandsvariable. Weil sie sich jedoch in ihrem Wesen so sehr von den Attributen unterscheiden, erhalten sie einen eigenen Platz im Deklarationsteil eingeräumt.

Bestände unterscheiden sich von den Attributen dadurch, daß ihr dynamisches Verhalten durch eine Transportanweisung und nicht durch eine Wertzuweisung beschrieben wird. Sie repräsentieren Mengen anstelle von Eigenschaften. Mehrere Komponenten können sich einen Bestand teilen, während ein Attribut stets genau einer Komponente zugeordnet ist.

Bestände sind disjunkte, geordnete Mengen und deshalb kann über sie auf die mobilen Elemente eines Modells eindeutig zugegriffen werden. Wir können vier verschiedene Zugriffsrechte auf Bestände unterscheiden:
– Leserecht
– Schreibrecht
– Bringerecht
– Holerecht.

Für Bestände mit gleichartigen Elementen entfällt die Möglichkeit zu schreiben.

Eine Komponente kann private Bestände oder zugängliche Bestände vereinbaren. Über private Bestände besitzt sie volle Autonomie, d.h sie besitzt alle Rechte, während andere Komponenten überhaupt keine Rechte mit Ausnahme des Leserechts besitzen (Exklusivrechte).

Zugängliche Bestände dienen dem Austausch von Elementen zwischen Komponenten. Mehrere Komponenten teilen sich einen Bestand und verfügen über gewisse Rechte, die einen Transport möglich machen.

Eine Komponente kann ihre Autonomie über einen Bestand teilweise abtreten, indem sie
- auf eigene Rechte verzichtet und/oder
- anderen Komponenten (ebenfalls) Rechte einräumt.

Je nachdem, wieviel eine Komponente von ihrer Autonomie abtritt, können Bestände mit unterschiedlichen Attributen deklariert werden.

Wir unterscheiden Bestände mit folgenden Rechten:

| | | | | | |
|---|---|---|---|---|---|
| private Bestände: | PRIVATE | L–WA | S–WK | B–WK | H–WK |
| Durchgangsbestände: | TRANSIT | L–WA | S–WA | B–WA | H–WA |
| Ausgangsbestände: | EXIT | L–WA | S–VA | B–WA | H–VA |
| Eingangsbestände: | ENTRY | L–WA | S–WK | B–VA | H–WK |
| verteilbare Eingangsbestände: | DENTRY | L–WA | S–WA | B–VA | H–WA |
| Quellen: | SOURCE | L–WK | S–WK | B–VK | H–WK |
| Senken: | SINK | L–WK | S–VK | B–WK | H–VK |
| Sensorbestände: | SENSOR | L–WA | S–VA | B–VA | H–VA |

Durchgangsbestande sind für alle frei zugänglich. Ausgangsbestände verzichten auf das Recht zu holen, und räumen anderen das Recht zu bringen ein. Dadurch können mehrere Ausgangsbestände zusammengeschlossen werden.

Eingangsbestände verzichten auf das Recht zu bringen. Verteilbare Eingangsbestände räumen auch anderen das Recht zu holen ein und können daher zusammengeschlossen werden und auch mit Durchgangsbeständen verbunden werden. “Einfache” Eingangsbestände nehmen das Recht zu holen exklusiv wahr und dürfen daher nur mit einer oder mehreren Ausgangsbeständen verbunden werden.

Quellen halten für den Benutzer stets eine unendliche Menge von Elementen vor, während Senken ankommende Elemente sofort aus dem Modell entfernen. Sensorbestände dürfen lesend auf fremde Bestände und deren mobile Elemente zugreifen.

```
location_declaration_part ::=

              [ PRIVATE  LOCATION[S]  loc_course_specific_declaration ]
              [ ENTRY    LOCATION[S]  loc_course_specific_declaration ]
              [ DENTRY   LOCATION[S]  loc_course_specific_declaration ]
              [ EXIT     LOCATION[S]  loc_course_specific_declaration ]
              [ TRANSIT  LOCATION[S]  loc_course_specific_declaration ]
              [ SENSOR   LOCATION[S]  loc_course_specific_declaration ]
              [ SOURCE   LOCATION[S]  loc_course_specific_declaration ]
              [ SINK     LOCATION[S]  loc_course_specific_declaration ]
```

Der Zeitverlauf eines Bestands kann diskret (sprunghaft) oder kontinuierlich (stetig) sein. In der Regel wird der Zeitverlauf diskret sein; ein konstanter Verlauf macht bei Beständen keinen Sinn.

```
loc_course_specific_declaration ::= [ [ DISCRETE ]  variable_declaration_list ]
                                    [  CONTINUOUS   variable_declaration_list ]
```

Die weitere Deklartion ist für Attribute und Bestände identisch.

```
variable_declaration_list ::=       variable_declaration
                             { ',' variable_declaration '}'

variable_declaration ::=
   [ARRAY  dimension_list] identifier '(' value_set ')'  [':=' initial_value ]
 | [ARRAY  dimension_list] identifier '(' value_set ')'  [':'  distribution  ]
```

Durch die Angabe von Dimensionen kann jede Variable als mehrdimensionales Feld vereinbart werden. Als Feldgrenzen sind konstante Zahlenwerte oder geklammerte Ausdrücke erlaubt, die sich bereits zur Übersetzungszeit berechnen lassen.

```
dimension_list   ::=  '{' index_range '}'  { '{' index_range '}' }

index_range      ::=  constant_integer '..' constant_integer

constant_integer ::=  operand
```

Die anzugebende Wertemenge kann um eine physikalische Maßeinheit ergänzt werden.

```
value_set ::=   INT        [ unit ]
              | REAL       [ unit ]
              | LOGICAL
              | identifer
              | SET OF identifier [ '||' ordering_criteria ]
```

Bestände, die individuelle Elemente enthalten, sind als Warteschlange organisiert. Ohne weitere Angaben gilt das FIFO-Prinzip. Bestände - mit Ausnahme von Quellen und Senken - können aber auch mit einem anderen Einordnungskriterium bzw. einer Liste von Kriterien versehen werden. Neben FIFO und LIFO, die nur am Ende der Liste stehen dürfen, kann nach aufsteigenden oder abfallenden Attributen der mobilen Komponenten in die Warteschlange einsortiert werden. Stehen nach Anwendung des ersten Kriterium noch mehrere Positionen für eine Einordnung zur Wahl, wird durch Anwendung des nächsten Kriteriums zwischen diesen entschieden. Wenn gar kein oder wenn keine weiteren Kriterien angegeben sind, wird nach FIFO entschieden.

Mehrere, in einer höheren Komponente zusammengeschlossene Bestände müssen die gleichen Einordnungskriterien besitzen.

```
ordering_criteria ::=  criterion  { ',' criterion }

criterion ::=   FIFO
              | LIFO
              | INC   identifier
              | DEC   identifier
              | TRUE  identifer
              | FALSE identifier
```

Zustandsvariablen und nicht angeschlossene Sensorvariablen, welche Attribute ausdrücken, sind mit einem Anfangswert zu versehen. Für alle anderen Attribute sowie für Bestände ist die Vorbesetzung optional. Fehlt die Anfangsbelegung, wird für Bestände ein leerer Bestand angenommen. Die Vorbesetzung von Quellen und Senken ist nicht möglich.

```
initial_value ::=   number [ unit | identifier ]
                  | TRUE
                  | FALSE
                  | " ' " identifier " ' "
```

Zufallsvariablen sind mit einer Verteilung zu versehen. Sie sind hier nur der Vollständigkeit halber mit aufgeführt und werden in Kapitel 6 kurz abgehandelt.

Auf die Einführung von abhängigen Mengenvariablen, die abhängigen Eigenschaftsvariablen entsprechen, wurde bewußt verzichtet. An dieser Stelle gibt es viele methodologische Probleme und ein praktischer Nutzen ist nicht erkennbar.

Vor allem gibt es kein Unterscheidungsmerkmal zwischen einer abhängigen Eigenschaftsvariablen und einer abhängigen Mengenvariablen, wenn es sich um gleichartige Elemente handelt, die nur durch eine Anzahl ausgedrückt werden.

## 5.3 Beschreibung der Modelldynamik in Basiskomponenten

Die Modelldynamik wird durch eine Folge von Aussageformen beschrieben. Der zeitliche Verlauf von Attributen wird durch Wertzuweisungen, der Verlauf von Beständen durch Transportanweisungen festgelegt.

Beide Arten von Aussagen können auch als bedingte Aussage oder als verallgemeinerte Aussage formuliert werden. Hierzu sind sie in ein IF- bzw. in ein FOREACH-Statement einzubetten.

```
dynamic_behaviour  ::=   DYNAMIC BEHAVIOUR
                         statement_sequence

statement_sequence ::=  statement { statement }

statement ::=  assignment
             | transport
             | conditional_statement
             | foreach_statement
```

Da der Zugriff auf Modellvariablen und die Bildung von Ausdrücken Teile aller Aussageformen und daher für das Verständnis unentbehrlich sind, seien zwei entsprechene Abschnitte vorangestellt.

### 5.3.1 Zugriff auf Modellvariablen

Modellvariablen gliedern sich in Attribute und Bestände. Zugriffe auf Attribute (selected_attribute) und Zugriffe auf Bestände (selected_location) werden aber in gleicher Weise formuliert. Variablen (selected_variables), welche auf einer mobilen Komponente plaziert sind, sind durch einen zweistufigen Selektionspfad über Bestand und mobiles Element zu erreichen.

```
selected_attribute ::=  selected_variable
selected_location  ::=  selected_variable

selected_variable  ::=   variable_path
                       | identifier '.' variable_path

variable_path      ::=   variable  { ':' element '.' variable }

element    ::=  identifier  index

variable   ::=  identifier  [ index_list ]

index_list ::=  index { index }

index      ::=  '[' index_expression ']'
```

Da jede Aussage in einen Allquantor eingebettet sein kann, kann der Zugriffspfad auf Variablen mobiler Komponenten sowohl mit einer Location als auch mit einem Bezeichner für eine gebundene Variablen beginnen, die für eine mobile Komponente steht.

Zugriffe auf einzelne mobile Elemente erfolgen über den gleichen Zugriffspfad, der dann aber mit einem Element endet.

```
selected_element ::=   selected_variable ':' element
                     | identifier
```

Ein ungelöstes Problem ist die Indexprüfung zur Compile-Zeit. Da dem Übersetzer nicht bekannt ist, ob und wieviele Elemente in der Warteschlange enthalten sind, werden Zugriffe auf nicht existierende mobile Elemente erst während des Programmlaufs festgestellt.

### 5.3.2 Die Formulierung von Ausdrücken

Es werden hier nur Ausdrücke betrachtet, die einen Wert liefern und nicht etwa eine Menge.

Bedingte Ausdrücke, d.h. Ausdrücke nach Fällen unterschieden, treten in Wertzuweisungen und in Transportanweisungen mit gleichartigen Elementen auf.

```
conditional_expression ::=   expression ';'
                           | expression  WHEN  expression
                           { ELSE
                             expression  WHEN  expression }
                           [ ELSE
                             expression ]
                             END
```

Einfache Ausdrücke finden als Bedingung in IF-Konstrukten und WHEN-Konstrukten sowie bei der Bildung von Eigenschaftsmengen Verwendung. Auch als Ordnungskriterium für Eigenschaftsmengen und zur Berechnung von Indices sind sie in Gebrauch.

Ein Ausdruck wird gebildet, indem Operanden durch zweistellige Operatoren miteinander verknüpft werden.

```
expression ::=  ['+' | '-']  operand  { op  operand }
```

Zu den Operanden zählen:

- Modellvariablen (Attribute und Bestände mit gleichartigen Elementen)
- Zahlenwerte und Dimensionierungskonstanten
- logische Werte (`TRUE` und `FALSE`)
- Werte aus Aufzählungsmengen
- Zeit `T`
- Funktionsaufrufe und einstellige Operatoren
- geklammerte Ausdrücke

Die zweistelligen Operationen lassen sich in numerische Rechenoperationen (`+`, `-`, `*`, `/`), logiche Rechenoperationen (`AND`, `OR`) und Vergleichsoperationen (`<`, `<=`, `=`, `<>`, `>`, `>=`) unterteilen.

Die numerischen Rechenoperationen und die Vergleichsoperationen lassen sich auf Attribute und Bestände mit gleichartigen Elementen in gleicher Weise anwenden.

Daneben liefert auch die zweistellige Operation

| | |
|---|---|
| `identifier  IN  cutted_set` | Element aus Menge ? |

an der eine Menge mit unterscheidbaren Elementen beteiligt ist, als Ergebnis einen logischen Wert.

Für den Vorrang bei der Verknüpfung der Operanden gelten die in der Mathematik üblichen Reglen (z.B. Punkt-vor-Strich).

Neben dem einstelligen Operator `NOT` haben vor allem auch die folgenden Operationen zur Abfrage von Mengeneigenschaften Bedeutung:

| | | |
|---|---|---|
| `CARD` | `set` | Kardinalität |
| `EMPTY` | `set` | Menge leer ? |
| `ALL` | `ordered_property_set` | Menge vollständig ? |
| `ANY` | `set` | Menge nicht leer ? |

### 5.3.3 Die Wertzuweisung

Die Wertzuweisung ordnet einem Attribut oder der Ableitung eines Attributs in Form einer Definitionsgleichung einen Wert zu. Dabei sind drei Fälle zu unterscheiden:

1. Einer abhängigen Variablen wird der für den aktuellen Zeitpunkt gültige Wert durch eine Definitionsgleichung zugeordnet.
2. Für eine Zustandsvariable wird beschrieben, wie sie sich zum folgenden Zeitpunkt gegenüber dem aktuellen Zeitpunkt verändert:
   (a) Ein diskreter Zustandsübergang legt den Wert des Attribut für den folgenden Zeitpunkt fest. Der Name des Attributs wird hierzu mit einem Häkchen `"^"` versehen.
   (b) Ein kontinuierlicher Zustandsübergang wird durch die im aktuellen Zeitpunkt gültige Steigung des Funktionsverlaufs beschrieben. Der Name des Attributs wird hierzu mit einem Apostroph `" ' "` versehen.

Die Wertzuweisung kann sich sowohl auf Attribute der Basiskomponente als auch auf Attribute mobiler Komponenten beziehen. Diese sind über ihren Aufenthaltsort zu erreichen.

```
assignment ::=  selected_attribute [tr_spec]  ':='  conditional_expression

tr_spec ::=    " ^ "
             | " ' "
```

### 5.3.4 Transporte zwischen Beständen

Schreibende Zugriffe auf Bestände können nur über Transportoperationen erfolgen. Es wird syntaktisch unterschieden, ob der Transport mit unterscheidbaren oder mit gleichartigen Elementen durchgeführt wird.

```
transport ::=   element_transport
              | quantity_transport
```

1. Transporte mit unterscheidbaren Elementen

   Transporte mit unterscheidbaren Elementen können nur mit einzelnen Elementen (elementweise) durchgeführt werden.

   ```
   element_transport ::=  selected_element  '->'  selected_location ';'
   ```

   Die Transportanweisung kann sich sowohl auf Bestände der Basiskomponente als auch auf Bestände mobiler Komponenten beziehen. Diese sind über ihren Aufenthaltsort zu erreichen.

2. Transporte mit gleichartigen Elementen

   Bei Transporten mit gleichartigen Elementen wird neben dem Transportpfad `(Bestand -> Bestand)` die zu transportierende Menge angegeben. Dabei ist zwischen diskreten und kontinuierlichen Transportvorgängen zu unterscheiden.

   b) Diskrete Transportvorgänge werden durch Angabe der Stückzahl beschrieben, die zum folgenden Zeitpunkt in den Zielbestand überwechselt.

   b) Kontinuierliche Transporte werden durch die im aktuellen Zeitpunkt gültige Transportrate beschrieben. Hierzu wird der Transportpfad mit einem Apostroph `" ’ "` versehen.

```
quantity_transport ::=

    ’(’ selected_location  ’->’  selected_location ’)’ [tr_spec]

                                    ’:=’  conditional_expression

tr_spec ::=   " ’ "
```

   Das Häkchen `"^"` als Übergangs-Spezifikator findet keine Verwendung.

### 5.3.5 Das IF-Konstrukt

Wertzuweisungen und Transportanweisungen können durch ein IF-Konstrukt nach Fällen unterschieden werden.

```
conditional_statement ::=   IF     expression  LET  statement_sequence
                          { ELSIF  expression  LET  statement_sequence }
                          [ ELSE               LET  statement_sequence ]
                            END
```

Bei der Anwendung dieses Konstrukts ist darauf zu achten, daß

- abhängige Variablen vollständig definiert werden, d.h. in allen Zweigen einschließlich des ELSE-Zweiges mit einer Definitionsgleichung vertreten sind
- alle Attribute und Bestände eindeutig definiert werden, d.h. außerhalb des IF-Konstrukts nicht mehr auftreten dürfen.

Bei der Verwendung indizierter Variablen gilt dies für jeden einzelnen Index.

### 5.3.6 Das FOREACH-Konstrukt

Das FOREACH-Konstrukt verallgemeinert Aussageformen für eine Menge von Indices. Hierzu stellt das FOREACH-Konstrukt einen Bezeichner (gebundene Variable) zur Verfügung, der für jeden einzelnen Index bzw. für jedes einzelne mobile Element einer angegebenen Menge steht.

Diese Menge kann angegeben sein

- durch einen Indexbereich, der durch konstante, ganze Zahlen festgelegt wird oder die aktuell belegten Indices einer Location umfaßt
- durch eine (beschnittene) Menge von Indices oder mobilen Elementen, die auch eine Eigenschaftsmenge sein darf.

Läßt man die Angabe einer Eigenschaftsmenge zu, können mobile Elemente aus Locations selektiert werden.

Die Aussageformen, die im Rumpf des Konstrukts stehen, sind indiziert und gelten für alle diese Indices.

```
foreach_statement    ::=   FOREACH  identifier  IN  range_or_set
                           LET   statement_sequence   END

range_or_set         ::=   actual_index_range
                         | cutted_set

actual_index_range  ::=   constant_integer  '..'  constant_integer
                        | IDXSET '(' selected_location ')'
```

Jede gebundene Variable muß dafür verwendet werden, um die im FOREACH-Konstrukt eingeschlossenen Aussagen zu quantisieren, d.h. sie muß die zu definierenden Variablen einer Wertzuweisung bzw. die Elemente oder den Quellbestand einer Transportanweisung näher bestimmen.

Für den Fall geschachtelter FOREACH-Konstrukte kann dies kann auch dadurch geschehen, daß eine gebundene Variable die Menge (`cutted_set`) eines eingeschlossenen FOREACH-Konstrukts näher bestimmt.

Beispiel:

```
FOREACH  i  IN  1 .. N
LET
    FOREACH  J  IN  Queue [i]
    LET
        J -> Server;
    END
END
```

Im Grunde wäre es ausreichend, nur Indices als gebundene Variable zuzulassen. Die Formulierung eines Modells wird jedoch wesentlich übersichtlicher, wenn die gebundene Variable auch mobile Komponenten repräsentieren kann.

### 5.3.7 Der implizite Allquantor

Häufig ist die Selektion von Variablen oder mobilen Elementen allein durch die Angabe von (evtl. eingeschränkten) Indexbereichen möglich. In diesen Fällen ermöglicht der implizite Allquantor eine besonders kompakte und übersichtliche Beschreibung. Methodologisch gesehen wäre er nicht erforderlich, da bereits das FORALL-Konstrukt alle seine Möglichkeiten mit einschließt.

Der implizite Allquantor ist sowohl auf Wertzuweisungen als auch auf Transportanweisungen anwendbar. Er besagt, daß eine solche Aussage nicht nur für einen einzigen Index, sondern für eine ganze Menge von Indices Gültigkeit besitzt.

```
index ::=   '[' index_expression  ']'
          | '{' implicit_indexset '}'

implicit_indexset ::=   index_range
                      | identifier  [ IN index_range ]  [ '|' condition ]
```

Bei

- zu definierenden Variablen in Wertzuweisungen
- Quellbeständen in Transportanweisungen
- Elementen aus Quellbeständen in Transportanweisungen mit unterscheidbaren Elementen

darf anstelle eines Indexausdrucks in eckigen Klammern eine Indexmenge in geschweiften Klammern stehen.

Die Angabe einer Indexmenge hinter einem Element entspricht dabei syntaktisch und semantisch dem Beschneiden eines Bestandes.

Auch die Angabe einer gebundenen Variablen ist möglich, welche alle Werte des angegebenen Indexbereichs annimmt. Diese Variable besitzt ihre Gültigkeit nur innerhalb dieser einen Anweisung. Fehlt die Angabe des Indexbereiches, dann nimmt die gebundene Variable alle möglichen Werte an. Für Elemente von Beständen bedeutet das, daß die gebundene Variable die Indices aller Elemente annimmt, die sich zur Zeit auf diesem Bestand befinden.

Die Indexmenge kann durch eine Bedingung, welche die gebundene Variable zu enthalten hat, eingeschränkt werden.

### 5.3.8 Beschneiden von geordneten Mengen

Das FOREACH-Konstrukt erlaubt es, bestimmte Operationen mit ausgewählten Mengenelemente durchzuführen. Es ist möglich, mit einer gesamten Menge zu operieren oder Mengenelemente aus einer geordneten Menge durch Beschneiden auszugrenzen.

Durch das Beschneiden wird eine geordnete Menge auf die Elemente mit denjenigen Positionen reduziert, die mit dem Indexbereich bzw. dem Index angegebenen sind.

```
cutted_set   ::=  set [ ':' [ identifier ]  cardinal_set ]

set          ::=   selected_location
                 | ordered_property_set

cardinal_set ::=   '{' constant_integer '}'
                 | '{' constant_integer .. constant_integer '}'
```

Es können sowohl Indexmengen wie auch Mengen von mobilen Elementen beschnitten werden. Im letzterem Fall ist der Klassenname der mobilen Komponenten anzugeben. Auf diese Weise wird eine Äquivalenz zum Zugriff von mobilen Elementen hergestellt, die sich auf Beständen befinden.

Als (geordnete) Menge kann

– ein Bestand (`location`) oder

– eine Eigenschaftsmenge (`ordered_property_set`)

herangezogen werden.

Eine Bildung von Ausdrücken, d.h. eine Verknüpfung von Mengen durch Operatoren ist an dieser Stelle nicht sinnvoll, weil dadurch die Mengen wieder ihre Ordnung verlören und somit nicht mehr beschnitten werden könnten. Ohne Einschränkung der Allgemeinheit läßt sich die Verknüpfung von Mengen hingegen bei der Bildung von Eigenschaftsmengen einbringen.

### 5.3.9 Die Bildung von Eigenschaftsmengen

Eigenschaftsmengen werden aus einer Grundmenge heraus gebildet. Die Grundmenge einer Indexmenge wird durch Angabe eines Indexbereiches, die Grundmenge einer Menge aus mobilen Elementen wird durch Angabe eines Bestandes oder einer Verknüpfung von Beständen angegeben.

Aus dieser Grundmenge kann durch die Angabe einer Bedingung eine reduzierte Menge gebildet werden, die durch Angabe eines oder mehrerer Ordnungskriterien noch in eine (andere) Ordnung gebracht werden kann. Ohne eine Neuordnung gilt die Ordnung der Grundmenge, bei Indexmengen eine aufsteigende Reihenfolge.

```
ordered_property_set ::=  '{' identifier  IN  basis_set
                              [ '|'  condition ] [ '||'  criteria_list ] '}'

basis_set ::=   actual_index_range
              | location_expression

actual_index_range ::=   constant_integer '..' constant_integer
                       | IDXSET '(' selected_location ')'

condition ::=   expression

criteria_list ::=  ordering_criterion { ',' ordering_criterion }

ordering_criterion ::=   INC    expression
                       | DEC    expression
                       | TRUE   expression
                       | FALSE  expression
```

Im Gegensatz zur Deklaration von Beständen, darf ein Ordnungskriterium hier ein Ausdruck sein, der beliebige Variablen des Modells miteinander verknüpft.

Da Bestände bereits mit einer Ordnung versehen wurden, erhält man aus einem durch eine Bedingung reduzierten Bestand wiederum eine geordnete Menge. Eine Neuordnung durch Angabe eines Kriteriums ist aber dennoch möglich.

### 5.3.10 Verknüpfen von Beständen mit unterscheidbaren Elementen

Für die Bildung von Eigenschaftsmengen sind Verknüpfungen

$$\text{Bestand} \times \text{Bestand} \quad \rightarrow \quad \text{Menge}$$

von Beständen mit unterscheidbaren Elementen interessant, die wiederum Mengen von unterscheidbaren Elementen erzeugen.

Als Verknüpfungsoperationen kämen die gängigen Mengenoperationen wie Vereinigung, Differenz, Durchschnitt und symmetrische Differenz in Frage. Da Bestände jedoch stets disjunkt sind, ist nur die Vereinigung von Beständen eine sinnvolle Verknüpfung.

```
location_expression ::=  selected_location { '+' selected_location }
```

Die Vereinigung wird durch den Additionsoperator `'+'` symbolisiert. Nur Bestände mit Elementen der gleichen Klasse können vereinigt werden.

Auf die Einführung einer derartigen Verknüpfung kann allerdings auch verzichtet werden, da sich ihr Gebrauch in der Praxis als unbedeutend erweist. Auch die im vorangegangenen Abschnitt besprochene Bildung von Eigenschaftsmengen, ist nicht unbedingt erforderlich, da die gleiche Funktionalität mit Funktionsaufrufen erreicht werden kann. Die Beschreibung durch Mengenbildung ist jedoch eleganter und der deklarative Teil der Sprache muß nicht verlassen werden.

## 5.4 Höhere Komponenten

Eine höhere Komponente setzt sich aus untergeordneten Komponenten (Subkomponenten) zusammen. Verbindungen zwischen Subkomponenten bestimmen die Struktur einer höheren Komponente und damit auch deren Verhalten.

Durch Vereinbarung eigener Variablen, die aus Variablen von Subkomponenten abgeleitet werden, kann eine Schnittstelle geschaffen werden, die es ermöglicht, die Komponente selbst wiederum als Subkomponente einzusetzen.

Die in den Subkomponenten durchgeführte Vorbesetzung kann in der höheren Komponente durch eine erneute Initialisierung überschrieben werden.

```
high_level_component ::=   HIGH LEVEL COMPONENT  identifier
                           subcomponent_declaration
                         [ interface_declaration ]
                         [ component_structure ]
                         [ initialization ]
                           END OF  identifier
```

### 5.4.1 Vereinbarung der Subkomponenten

Eine höhere Komponente setzt sich aus untergeordneten Komponenten (Subkomponenten) zusammen. Die Subkomponenten sind Ausprägungen (Instanzen, Inkarnationen) bereits definierter Komponentenklassen (Klassenkonzept). Ausprägungen der gleichen Klasse können durch eigene Namen und/oder – bei Feldern von Subkomponenten – durch Indices unterschieden werden.

```
subcomponent_declaration ::=  SUBCOMPONENT[S]
                              subcomponent { ',' subcomponent }

subcomponent ::=  [ ARRAY dimension_list ] identifier [ OF CLASS identifier ]
```

Felder von Komponenten werden genauso angelegt wie Felder von Variablen.

### 5.4.2 Einrichten einer Schnittstelle

Um eine höhere Komponente wiederum als Subkomponente einsetzen und damit Modelle hierarchisch aufbauen zu können, ist es möglich, eine Schnittstelle einzurichten. Zu diesem Zweck werden in der höheren Komponente eigene Variablen (Attribute und Locations) vereinbart, denen Variablen aus Subkomponenten entsprechen. Beliebige Variablen und Bestände aus Subkomponenten – mit Ausnahme der privaten Bestände – können in einer höheren Komponente mit eigenen Namen belegt werden.

```
interface_declaration ::=  DECLARATION[S]
                          [ declaration_of_sensor_variables ]
                          [ declaration_of_internal_variables ]
                          [ declaration_of_entry_locations ]
                          [ declaration_of_dentry_locations ]
                          [ declaration_of_exit_locations ]
                          [ declaration_of_transit_locations ]

declaration_of_sensor_variables   ::=  SENSOR VARIABLE[S]
                                      { sensor_variable_declaration }

declaration_of_internal_variables ::=  INTERNAL VARIABLE[S]
                                      { internal_variable_declaration }

declaration_of_entry_locations    ::=  ENTRY LOCATION[S]
                                      { entry_location_declaration }

declaration_of_dentry_locations   ::=  DENTRY LOCATION[S]
                                      { dentry_location_declaration }

declaration_of_exit_locations     ::=  EXIT LOCATION[S]
                                      { exit_location_declaration }
```

```
declaration_of_transit_locations  ::=   TRANSIT LOCATION[S]
                                       { transit_location_declaration }
```

Der Wert einer Sensorvariablen kann an eine oder auch mehrere Sensorvariablen aus Subkomponenten weitergereicht werden.

```
sensor_variable_declaration  ::=
      identifier  '-->'  sensor_variable ';'
    | identifier  '-->' '(' sensor_variable { ',' sensor_variable ')' ';'
```

Der Wert einer internen Variablen kann von einer Zustandsvariablen oder abhängigen Variablen einer Subkomponente übernommen werden.

```
internal_variable_declaration  ::=
      identifier  '<--'  internal_variable ';'

sensor_variable   ::=  indexed_component '.' indexed_variable
internal_variable ::=  indexed_component '.' indexed_variable

indexed_component ::=  indexed_identifier
indexed_variable  ::=  indexed_identifier
```

Einen Eingangsbestand für eine höhere Komponente erhält man durch Gleichsetzung mit einem (nicht verteilbaren) Eingangsbestand einer Subkomponente.

```
entry_location_declaration  ::=   identifier  '==>'  entry_location ';'
                                | identifier  '==>'  dentry_fusion ';'

dentry_fusion  ::=   dentry_location
                   | '(' dentry_location  {  '=='  dentry_location  } ')'
```

Alternativ kann ein Eingangsbestand einer höheren Komponente auch durch Gleichsetzung mit einem Zusammenschluß mehrerer verteilter Eingangsbestände (DENTRY) vereinbart werden.

Einen verteilbaren Eingangsbestand (DENTRY) für eine höhere Komponente erhält man durch Gleichsetzung mit einem (verteilten oder nicht verteilten) Eingangsbestand einer Subkomponente oder durch Gleichsetzung mit einem Zusammenschluß mehrerer verteilter Eingangsbestände (DENTRY).

```
dentry_location_declaration ::=   identifier  '==>'  dentry_location ';'
                                | identifier  '==>'  dentry_fusion ';'
```

Einen Ausgangsbestand für eine höhere Komponente erhält man durch Gleichsetzung mit einem Ausgangsbestand einer Subkomponente oder durch Gleichsetzung mit einem Zusammenschluß mehrerer Ausgangsbestände.

```
exit_location_declaration ::=   identifier  '<=='  exit_location ';'
                              | identifier  '<=='  exit_fusion ';'

exit_fusion    ::=   exit_location
                   | '(' exit_location    {  '=='  exit_location    } ')'
```

Einen Durchgangsbestand für eine höhere Komponente erhält man durch Gleichsetzung mit einem Durchgangsbestand einer Subkomponente oder durch Gleichsetzung mit einer Transportverbindung zwischen Subkomponenten.

```
transit_location_declaration
                        ::=   identifier  '<==>'  transit_fusion ';'
                            | identifier  '<==>'  transport_connection ';'

transit_fusion ::=   transit_location
                   | '(' transit_location { '<==>' transit_location } ')'
```

Eine Transportverbindung beinhaltet mindestens den Zusammenschluß zweier Durchgangsbestände oder eine Verbindung zwischen einem Ausgangs- und einem (verteilbaren) Eingangsbestand.

```
entry_location    ::=  indexed_component '.' indexed_location
dentry_location   ::=  indexed_component '.' indexed_location
exit_location     ::=  indexed_component '.' indexed_location
transit_location  ::=  indexed_component '.' indexed_location

indexed_component ::=  indexed_identifier
indexed_location  ::=  indexed_identifier
```

Beispiele hierzu finden sich im Kapitel 4.6 .

### 5.4.3 Verbindungen zwischen Komponenten

Der strukturelle Aufbau einer höheren Komponente wird durch Verbindungen zwischen Komponenten dargestellt. Verbindungen zwischen Komponenten können sowohl Wirkungen von Attributen auf andere Komponenten zum Ausdruck bringen als auch Zusammenschlüsse von Beständen repräsentieren, die dem Transport von Elementen dienen.

```
component_structure ::=   STRUCTURE
                        [ effect_connection_part    ]
                        [ transport_connection_part ]
```

Wirkungen gehen von internen Variablen (Zustandsvariablen oder abhängigen Variablen) einer Komponente aus und nehmen über Sensorvariablen auf die Dynamik einer oder mehrerer anderer Komponenten Einfluß.

```
effect_connection_part ::=   EFFECT CONNECTION[S]
                            { effect_connection }

effect_connection ::=
    internal_variable '-->' sensor_variable ';'
  | internal_variable '-->' '(' sensor_variable
                                 { ',' sensor_variable ')' } ';'

internal_variable ::=   indexed_component '.' indexed_variable
sensor_variable   ::=   indexed_component '.' indexed_variable

indexed_component ::=  indexed_identifier
indexed_variable  ::=  indexed_identifier
```

Transportverbindungen sind Zusammenschlüsse von Beständen, die dem Austausch von (gleichartigen oder unterscheidbaren) Elementen zwischen Komponenten dienen.

Ein Ausgangsbestand oder ein Zusammenschluß von Ausgangsbeständen darf

- mit einer einzigen ENTRY-Location oder
- mit einer oder mehreren DENTRY- bzw. TRANSIT-Locations

verbunden sein.

Ein verteilbarer Eingangsbestand (DENTRY) oder ein Zusammenschluß von solchen darf mit einer oder mehreren EXIT- bzw. TRANSIT-Locations verbunden sein.

Mehrere zusammengeschlossene Transit-Locations definieren ebenfalls eine Transportverbindung.

Im Grunde sind alle Verbindungen möglich. Es ist nur darauf zu achten, daß Zusammenschlüsse von EXIT- oder DENTRY-Locations nicht für sich alleine stehen.

```
transport_connection_part ::=   TRANSPORT CONNECTION[S]
                               { transport_connection }


transport_connection ::=   exit_fusion '==>'  entry_location ';'

                         | exit_fusion '==>'  dentry_fusion ';'

                         | exit_fusion  '==>'  transit_fusion ';'

                         | exit_fusion  '==>'  transit_fusion
                                        '==>'  dentry_fusion  ';'

                         | transit_fusion  '==>'  dentry_fusion  ';'

                         | transit_location  '<==>' transit_location
                                           { '<==>' transit_location }
```

```
exit_fusion    ::=   exit_location
                   | '(' exit_location    {  '=='  exit_location    } ')'

dentry_fusion  ::=   dentry_location
                   | '(' dentry_location  {  '=='  dentry_location  } ')'

transit_fusion ::=   transit_location
                   | '(' transit_location { '<==>' transit_location } ')'

exit_location     ::=  indexed_component '.' indexed_location
entry_location    ::=  indexed_component '.' indexed_location
dentry_location   ::=  indexed_component '.' indexed_location
transit_location  ::=  indexed_component '.' indexed_location

indexed_component ::=  indexed_identifier
indexed_location  ::=  indexed_identifier
```

Die Verwendung indizierter Komponenten, Variablen und Bestände macht auch bei der Formulierung von Verbindungen in höheren Komponenten die Verwendung eines expliziten oder impliziten Allquantors möglich. Dadurch wird die Strukturbeschreibung großer Komponenten mit regelmäßigem Aufbau stark vereinfacht.

### 5.4.4 Initialisierung der Subkomponenten

Die Anwendung des Klassenkonzepts gestattet den mehrfachen Einsatz der gleichen Modellkomponente. Die Vorbesetzung, die bei der Definition einer Komponentenklasse angegeben ist, gilt demnach für alle Ausprägungen einer Klasse. Eine individuelle Vorbesetzung kann in der höheren Komponente vorgenommen werden.

Die in den Subkomponenten durchgeführte Vorbesetzung der Attribute kann mit einem neuen Wert überschrieben werden.

Die Bestände können mit einer veränderten Anzahl mobiler Komponenten vorbesetzt werden. Die Vorbesetzung von Beständen ist immer dann notwendig, wenn durch eine Zusammenlegung von Beständen widersprüchliche Initialisierungen gelten. In diesem Fall ist die Vorbesetzung für alle zusammengelegten Bestände durchzuführen. Aber auch private Bestände können initialisiert werden.

```
initialization ::=   INITIAL CONDITION[S]
                   { variable_initialization  |  location_initialization }

variable_initialization ::=   variable  ':='  expression ';'
location_initialization ::=   fusion    ':='  number  identifier ';'

fusion ::=   location
           | '('  location  { '==' location }  ')'
           | transport_connection
```

```
variable ::=  indexed_component '.' indexed_variable
location ::=  indexed_component '.' indexed_location

indexed_component ::=  indexed_identifier
indexed_location  ::=  indexed_identifier
```

Beispiel:

```
INITIAL CONDITIONS
A.X     :=  15;
A.Queue :=  5 Job;
(A.Queue == B.Queue) := 1 Job;
A.Server ==> C.Queue := 5 Job;
```

# Kapitel 6

# Spracherweiterungen für den praktischen Einsatz

## 6.1 Anforderungen

Das bisher besprochene Sprachkonzept bildet die Grundlage für die zustandsorientierte Beschreibung von Modellen. Für einen praktischen Einsatz sind diese Konstrukte – trotz des bisher erreichten Sprachumfangs – jedoch noch nicht ausreichend.

Hierin zeigt sich auch das Dilemma einer Spezifikationssprache:

1. Der Anwender ist bei der Formulierung seines Modells auf die angebotenen sprachlichen Mittel angewiesen.

   Beispiele:
   - Wenn die Sprache keine Zufallsvariablen anbietet, kann der Anwender keine stochastischen Modelle formulieren.
   - Wenn die Sprache keine Differentialgleichungen zuläßt, können keine kontinuierlichen Modelle formuliert werden.
   - Wenn die Sprache nur arithmetische Ausdrücke zur Formulierung von Zusammenhängen zwischen Modellvariablen kennt, können Funktionen nicht tabellarisch oder algorithmisch dargestellt werden.

2. Der Anwender kann Funktionen, die über die reine Modellbeschreibung hinausgehen, nicht in das erzeugte Simulationsprogramm aufnehmen.

   Beispiele:
   - Laufende Ein/Ausgabe von Daten während eines Simulationslaufs von/auf Datei, Datenbank oder Bildschirm
   - Kommunikation mit anderen Prozessen während des Simulationslaufs
   - Synchronisation mit der Systemuhr (Echtzeit- bzw. zeitproportionaler Betrieb)

Eine prozedurale Sprache kennt diese Probleme nicht. Jede Art von Funktionalität, die den Umfang der Sprache übersteigt, kann in Form einer Unterprogrammsammlung zur Verfügung gestellt werden. In einem erheblichen Umfang kann der Anwender sich Unterprogramme mit

den Mittel einer Programmiersprache selbst erstellen. Lediglich Funktionen, die auf die Hardware oder das Betriebssystem des Rechners zurückgreifen, müssen als Systemprogramme bereits zur Verfügung stehen.

Bei einer Spezifikationssprache wie der diskutierten Modellbeschreibungssprache muß man sich nun überlegen, in welcher Form man die gewünschten Erweiterungen zur Verfügung stellt.

Wünsche nach einer erweiterten Funktionalität lassen sich in drei Kategorien unterscheiden:

1. Erweiterungen, die in den Compiler eingehen müssen:
   - Aufzählungsmengen
   - physikalische Maßeinheiten
   - Funktions- und Prozeduraufrufe zur Beschreibung komplexerer Zusammenhänge zwischen Modellvariablen (z.B. Formulieren von Strategien)
2. Erweiterungen, die sich durch Unterprogramme abdecken lassen aber sehr häufig genutzt werden und Teil der Modellbeschreibung sind:
   - Zufallsvariablen
   - Tabellenfunktionen
3. Erweiterungen, die nicht Teil der Modellbeschreibung sind und sich durch Unterprogramme abdecken lassen:
   - Laufende Ein/Ausgabe von Daten während eines Simulationslaufs von/auf Datei, Datenbank oder Bildschirm
   - Kommunikation mit anderen Prozessen während des Simulationslaufs
   - Synchronisation mit der Systemuhr (Echtzeit- bzw. zeitproportionaler Betrieb)

Die Wünsche aus der ersten Gruppe müssen selbstverständlich in die Sprache eingehen, da sie vom Compiler zu berücksichtigen sind. Bei Wünschen aus der zweiten Gruppe kann individuell abgewogen werden, ob und in welcher Weise sie Eingang in die Sprache finden sollen. Bei den Wünschen aus der dritten Gruppe fällt auf, daß sie häufig von Systemprogrammen Gebrauch machen. Hierzu ist es erforderlich, eine Schnittstelle zur Basissprache zu schaffen.

Die folgenden Abschnitte deuten an, wie diesen Wünschen der Anwender Rechnung getragen werden kann. Es soll damit das Sprachkonzept abgerundet werden, so daß keine Einschränkungen an den praktischen Einsatz gemacht werden müssen.

Von besonderer Bedeutung hierfür ist die Ausgestaltung einer Unterprogramm-Schnittstelle. Diese ermöglicht es dem Anwender, sowohl Funktionen und Prozeduren zu schreiben, die vom Compiler der Modellbeschreibungssprache bearbeitet werden als auch Unterprogramme in der Implementierungssprache C einzubinden. Im ersten Fall braucht der Anwender die Sprachumgebung nicht verlassen und bekommt vom Compiler eine umfangreiche Schnittstellenprüfung geboten. Im zweiten Fall hat der Anwender die Möglichkeit, Systemprogramme aufzurufen oder sich Schnittstellen zu anderen Programmen und Datenbeständen einzurichten.

Gleichzeitig verhindert die Unterprogramm-Schnittstelle, daß die Modellbeschreibungssprache durch immer neue Konstruktionen erweitert werden muß. Bei Anforderungen, die über die Funktionalität der Sprache hinausgehen, kann sich der Anwender selber weiterhelfen.

Die folgenden Beschreibungen sollen nur einen Überblick vermitteln und so viel Verständnis schaffen, wie es für die im Anschluß folgenden Anwendungen nötig ist. Eine detaillierte Beschreibung findet sich im Referenzhandbuch der Modellbeschreibungssprache SIMPLEX-MDL [Esch 91].

## 6.2 Aufzählungsmengen

Neben den standardmäßig angebotenen Wertemengen INTEGER, REAL und LOGICAL sind insbesondere benutzerdefinierte Aufzählungsmengen einer besseren Lesbarkeit des Modells dienlich. Neben ungeordneten Aufzählungen sind auch geordnete Aufzählungen denkbar.

Beispiel:

```
DEFINITIONS
   VALUE SET  Status:  ( 'wartet', 'arbeitet', 'gestoert' )
   VALUE SET  Groesse: ( 'klein' < 'mittel' < 'gross' )
```

Zur Unterscheidung von benutzerdefinierten Variablen- und Komponentennamen werden Ausprägungen einer Aufzählungsmenge in einfache Hochkommas eingeschlossen.

Werte von Aufzählungsmengen dürfen in mehreren Mengen enthalten sein. Dadurch darf es jedoch keine Widersprüche in der Ordnungsrelation geben.

Die Deklaration von Variablen erfolgt unter Angabe der Aufzählungsmenge.

Beispiel:

```
STATE VARIABLE
DISCRETE        Zustand (Status) :=  'wartet'
```

An Operationen ist bei ungeordneten Mengen die Prüfung auf Gleichheit (`'='`) oder Ungleichheit (`'<>'`) möglich, bei geordneten Mengen sind auch die übrigen Relationen erlaubt.

Der Name des Aufzählungstyps mitsamt den Ausprägungen wird in die darüberliegenden Komponenten vererbt. Dadurch wird eine Initialisierung ohne erneute Definition möglich und es läßt sich sicherstellen, daß eine in mehreren Komponenten mit gleichem Namen vereinbarte Aufzählungsmenge im gesamten Modell identisch definiert ist. Dies trägt zur Übersichtlichkeit bei und ist für den Fall wichtig, daß eine Sensorvariable über eine Verbindung an eine Variable in einer anderen Komponente angeschlossen ist.

## 6.3 Physikalische Maßeinheiten

Modellvariablen sind in vielen Anwendungen mit Einheiten behaftet. Eine Kontrolle der Einheiten bei Verknüpfungen von Modellvariablen erleichtert den Modellaufbau ungemein und ist von großem dokumentarischen Wert. Deshalb wird in SIMPLEX-MDL standardmäßig das SI-Einheitensystem mit den genormten Einheiten und Vorfaktoren zur Verfügung gestellt. Darüber hinaus wird dem Anwender die Möglichkeit gegeben, eigene Einheiten oder gar ein eigenes Einheitensystem zu definieren.

Bei der Deklaration einer Variablen wird die Einheit zusammen mit der Wertemenge angegeben.

Beispiel:
```
STATE VARIABLE
CONTINUOUS       X (REAL [m])  :=  10 [m]
```

Zur Unterscheidung von anderen Bezeichnern werden Einheiten in eckige Klammern gesetzt.

Wird mit Einheiten gearbeitet, ist für die Standardvariable T ebenfalls eine Einheit anzugeben.

Beispiel:
```
TIMEUNIT = [min]
```

Mit diesen Informationen lassen sich nun bei der weiteren semantischen Analyse eine Vielzahl von Prüfungen durchführen:

Bei Operationen in Ausdrücken wird bei Summationen und Vergleichen auf Identität der Einheiten und geprüft und bei einer Produktbildung eine neue Einheit als Produkt gebildet.

Beispiel:
```
X + Y       # Y muss ebenfalls die Einheit [m] besitzen
X / V       # Besitzt V die Einheit [m/s], dann resultiert
            # aus der Division die Einheit [s]
```

Für den Fall, daß zwei Einheiten zwar nicht identisch, aber ineinander überführbar (kommensurabel) sind, muß die Möglichkeit bestehen, die Einheit eines Ausdrucks, Teilausdrucks oder einer Variable in eine andere Einheit umzuwandeln.

Beispiel:
```
(X / V) [min]     # Der Ausdruck mit der Einheit [s] wird in
                  # die Einheit [min] umgewandelt, d.h.
                  # implizit durch 60 dividiert.
```

Bei Wertzuordnungen muß ebenfalls die Identität der Einheiten hergestellt werden. Dies kann explizit durch den Benutzer oder implizit durch den Compiler erfolgen, da bekannt ist, welche Einheit benötigt wird.

Beispiel:
```
TWeg :=  (X/V) [min];   # Besitzt TWeg die Einheit [min], ist
                        # eine Anpassung der Einheit notwendig.
TWeg :=  X / V;         # Implizite Anpassung
```

Identität der Einheiten ist ebenfalls erforderlich, wenn Variablen aus verschiedenen Komponenten miteinander verbunden werden.

Beispiel:
```
A.X  -->  B.X;      # Die Variable X aus Komponente B muss
                    # dieselben Einheiten besitzen wie Variable Y
                    # aus Komponente A.
```

Wenn Modelle aus Komponenten zusammengesetzt werden, die von verschiedenen Benutzer erstellt wurden, wirkt sich diese verbesserte Schnittstellenprüfung besonders positiv aus.

## 6.4 Funktions- und Prozeduraufrufe

Zusammenhänge zwischen Modellvariablen lassen sich nicht immer in geschlossenen Ausdrücken (ggf. auch mit Fallunterscheidungen) oder tabellarisch darstellen. Insbesondere bei der Formulierung von Strategien, wo eine Auswahl aus mehreren Alternativen zu treffen ist, sind auch Schleifenkonstuktionen notwendig.

Beispiel: Dieser Algorithmus sucht nach der Strategie `SCAN` aus einer Warteschlange `Queue` bei gegebener Position `ActPos` den in der aktuellen Richtung nächstgelegenen Auftrag (`PosDirec = TRUE | FALSE`). Der größtmögliche Abstand bei `N` Positionen ist `(N-1)`.

```
FUNCTION Scan (SET OF Job: Queue, INT: ActPos, LOGICAL: PosDirec  -->  INT)
DECLARE
         MinDist (INT), N (INT), IMin (INT)
BEGIN
    N       := 80;
    IMin    :=  0;
    MinDist :=  N;

    FOR i FROM 1 TO CARD(Queue)
    REPEAT
       IF  PosDirec
       DO
           IF  Queue: Job[i]. Pos >= ActPos
           DO
               Dist :=  Queue: Job[i]. Pos - ActPos;
           END
           ELSE
           DO
               Dist :=  (ActPos - Queue: Job[i]. Pos) + (N - ActPos);
           END
       END
       ELSE
           IF  Queue: Job[i]. Pos <= ActPos
           DO
               Dist :=  ActPos - Queue: Job[i]. Pos;
           END
           ELSE
           DO
               Dist :=  (Queue: Job[i]. Pos - ActPos) + (ActPos - 1);
           END
       END

       IF  Dist < MinDist
       DO
           MinDist := Dist;    IMin := i;
       END

    END_LOOP
```

```
     RETURN (IMin);

  END_FUNC
```

Das Beispiel zeigt folgendes:

- Die beschriebene Strategie läßt sich mit den bisherigen sprachlichen Mitteln nicht mehr beschreiben.
- Prozedurale Formulierungen benötigen den gesamten Sprachumfang wie er in gewöhnlichen Programmiersprachen zur Verfügung gestellt wird:
  - Deklaration lokaler Variablen
  - Wertzuweisungen
  - Fallunterscheidungen
  - Schleifen
- Besonderes angenehm für den Anwender ist es, daß er Locations als Eingabeparameter übergeben kann und die ihm bereits vertrauten dieselben Zugriffmöglichkeiten auf mobile Elemente und deren Attribute hat.

Auch wenn dieser Aufwand sehr hoch erscheint, er ermöglicht es auf der anderen Seite den Sprachumfang im deklarativen Teil in Grenzen zu halten. Anderfalls müßte die deklarative Sprache für Spezialfälle wie für das gezeigte Beispiel jedesmal entsprechend erweitert werden.

Selten benötigte Sprachkonstrukte werden so vermieden und die Universalität der Sprache gewährleistet.

Die Schnittstelle wurde so gehalten, daß ein Unterprogrammaufruf frei ist von jeglichen Seiteneffekten. Dies läßt sich insbesondere deshalb erreichen, weil die Sprache keine globalen Variablen kennt. Alle benötigten Parameter sind in der Parameterliste zu übergeben.

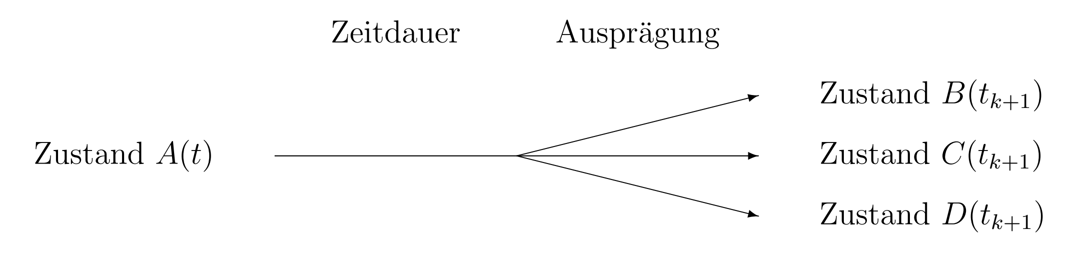
## 6.5 Zufallsvariablen

Der Übergang einer Zustandsvariablen auf einen neuen Wert (Zustandsübergang) kann mit Sicherheit (deterministisch) stattfinden oder aber mit einer gewissen Wahrscheinlichkeit (stochastisch) erfolgen. Dabei kann

a) die Zeitdauer, in der ein Zustand verweilt und/oder

b) die folgende Ausprägung (Wert) der Zustandsvariable

zufallsabhängig sein.



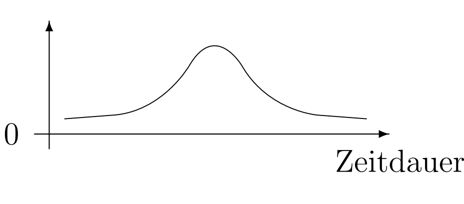
Jede zufällig zu ermittelnde Zeitdauer bzw. Ausprägung ist eine Zufallsvariable. Diese werden durch Verteilungs- bzw. Verteilungsdichtefunktionen beschrieben, welche aus empirisch bestimmten Häufigkeiten abgeleitet werden.

Beispiele:

| Zufallvariable | Verteilungsdichte |
|---|---|

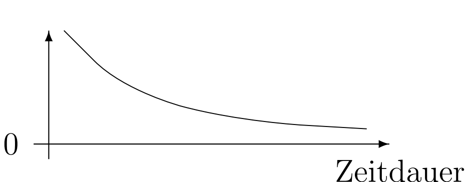
Dauer eines Bedienvorgangs

```
'arbeitet --> 'fertig'
```

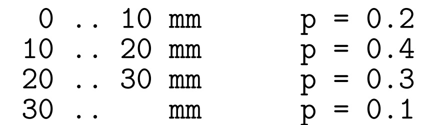



Dauer bis zur nächsten Ankunft

```
'wartet --> 'Ankunft'
```




Niederschlagsmenge bei Regen







Zufallsvariable können wie in den Beispielen statistisch unabhängig sein, aber auch mit vorangegangenen Werten korreliert und/oder mit anderen Zustandsvariablen korreliert sein. Je nachdem werden ein- oder mehrdimensionale Verteilungsfunktionen benötigt.

Ein neuer Wert einer Zufallsvariable wird mit Hilfe einer Zufallszahlenfolge bestimmt, die auf die gegebenen Verteilungsfunktionen angewendet werden.

Beispiele:

Es seien $x$, $x_1$ und $x_2$ Zufallsvariable; $r$, $r_1$ und $r_2$ seien Zufallszahlenfolgen aus $[0,1]$; $F_1$ sei eine eindimensionale, $F_2$ eine zweidimensionale Verteilungsfunktion.

3) Eine Zufallsvariable, statistisch unabhängig:
   $x(k) = F_1^{-1}(r(k))$
3) Eine Zufallsvariable, Korrelation zum vorangegangenen Wert:
   $x(k) = F_2^{-1}(r(k), x(k-1))$
3) Zwei Zufallsvariablen, Korrelation untereinander, keine Korrelation zum vorangegangenen Wert:
   $x_1(k) = F_1^{-1}(r_1(k))$
   $x_2(k) = F_2^{-1}(r_2(k), x_1(k))$

Zufallsvariable sind demnach unter Angabe einer Verteilungsfunktion, einer Zufallszahlenfolge und der korrelierten Variablen zu deklarieren. In SIMPLEX-MDL wurden bislang nur statistisch unabhängige Zufallsvariablen realisiert.

Beispiel:

```
RANDOM VARIABLE  TWork (REAL):  GAUSS (Mean := 10, Sigma := 3)
```

Eine bestimmte Zufallszahlenfolge wird vom Compiler zugeteilt. Eigentlich könnte man die Zufallszahlen auch einer einzigen (idealen) Zufallszahlenfolge entnehmen. Dies hätte aber den Nachteil, daß die einzelnen Prozesse nach Änderungen am Modell (auch der Parameter) wegen Verschiebungen im zeitlichen Ablauf nicht mehr das gleiche zufällige Verhalten aufweisen würden. Dadurch ist die Vergleichbarkeit von Simulationsergebnisse nicht mehr gegeben.

Die Integration von Zufallsvariablen in die Dynamikbeschreibung einer zustandsorientierten Sprache bereitet bislang noch methodologische Schwierigkeiten.

Im Gegensatz zu den Modellvariablen besitzen Zufallsvariablen nur vorübergehend einen Wert. In anderen Zeitabschnitten sind sie undefiniert. Der Wert einer Zufallsvariable, die eine Zeitdauer ausdrückt, ist nur interessant, solange ein bestimmter Zustand vorliegt. Wird dieser Zustand verlassen, ist der Wert der Zufallsvariable ohne Bedeutung. Zufallsvariablen, die einer Ausprägung entsprechen, werden sogar immer nur für einen Augenblick benötigt.

Unstrittig ist, daß eine Zufallsvariable zu einem Zeitpunkt nur einen Wert annehmen kann. Hingegen ist offen, wodurch eine Zufallsvariable veranlaßt werden soll, ihren Wert zu ändern. Drei Alternativen stehen zur Auswahl:

1. Eine Anweisung in einem Ereignis, z.B.

   ```
   DRAW  TWork;
   ```

   veranlaßt das Ziehen einer neuen Zufallszahl, die dann im nächsten Zeitpunkt (bzw. Moment) zur Verfügung steht.

   Nachteil: Der neue Wert der Zufallsvariablen wird meist bereits schon zum aktuellen Zeitpunkt benötigt.

2. Ein Funktionsaufruf, z.B.

   ```
   NXTRAN (TWork)
   ```

   erlaubt bereits den Zugriff auf den nächsten Wert der Zufallszahlenfolge und veranlaßt, daß TWork vom nächsten Zeitpunkt (bzw. Moment) an den neuen Wert besitzt.

   Nachteil: Zu einem Zeitpunkt besteht Zugriffsmöglichkeit auf den alten wie den neuen Wert der Zufallsvariablen.

3. Zufallsvariable dürfen nur in Ereignisfunktionen verwendet werden. Jeder Zugriff auf die Zufallsvariable in einer Wertzuordnung an eine Zustandsvariable, z.B.

   ```
   TReady^ := T + TWork;
   ```

   liefert einen für den aktuellen Zeitpunkt neuen Wert zurück.

   Nachteil: Kein informativer Zugriff möglich, da sich dadurch der Variablenwert verändert.

Keine der drei Alternativen kann vollständig zufriedenstellen. In SIMPLEX-MDL wurde zunächst einmal die Alternative drei realisiert. Das folgende Beispiel zeigt, daß diese Alternative eine gute Lesbarkeit sicherstellt.

Beispiel: Modellierung eines binären Zufallsprozesses

```
STATE VARIABLES   X       (LOGICAL) :=  FALSE ,
                  TChange (REAL)    :=  0

RANDOM VARIABLE   TDtrue  (REAL):  GAUSS (Mean := 10, Sigma := 3) ,
                  TDfalse (REAL):  EXPO  (Mean := 50)

IF  T >= TChange
LET
    IF  X = FALSE
    LET
        X^ := TRUE;
        TChange^ :=  T + TDtrue;
    END
    ELSE
    LET
        X^ := FALSE;
        TChange^ :=  T + TDfalse;
    END
END
```

In der Praxis hat sich diese Lösung auch durchaus bewährt.

## 6.6 Tabellarische Funktionen und Verteilungen

Zusammenhänge in Modellen, die auf empirisch ermittelten Daten, also nicht auf einer bereits vorhandenen Theorie beruhen, lassen sich zunächst vielfach nur durch tabellarische Funktionen beschreiben. Zwischen den bekannten Stützwerten ist ggf. noch in geeigneter Weise zu interpolieren.

Diese Tabellenfunktionen können recht umfangreiche Datenbestände ausmachen. Es erscheint daher nicht zweckmäßig, diese Tabellen mit in die Modellbeschreibung aufzunehmen, zumal sie meist bereits auf externen Dateien vorliegen.

Andererseits ist die Beschreibung eines Zusammenhangs zwischen Modellvariablen Bestandteil eines Modells und sollte daher nicht im verborgenen bleiben. Das sieht man auch daran, daß man ein Modell nicht an einen anderen Anwender weitergeben kann, ohne ihm die Funktionstabellen mitzugeben.

Aus diesem Grund wurden tabellarische Funktionen und Verteilungsfunktionen mitsamt ihrer Stützstellen und Stützwerte in den Modelltext aufgenommen.

Es ist zu unterscheiden, ob Tabellenfunktionen ein-, zwei- oder mehrdimensional sind und ob sie einen kontinuierlichen oder diskreten Verlauf beschreiben.

Kontinuierliche Verläufe sind zwischen den Stützstellen zu interpolieren. Dabei hat es sich als ausreichend erwiesen, vier verschiedene Arten der Interpolation anzubieten:

– stufenförmige,
– lineare,
– kubische Bessel,
– kubische Spline Interpolation

Mit zunehmender Reihenfolge wird im Kurvenverlauf eine weitere Ableitung stetig.

Die Verwendung von Tabellenfunktionen mit mehr als zwei Eingabeparametern ist sehr ungewöhnlich. Damit das Datenmaterial nicht zu umfangreich wird, zieht man es in der Regel vor, in einem solchen Fall mit approximierenden Funktionen zu arbeiten, auch wenn diese oft nicht sonderlich genau sind. Aus diesem Grund erscheinen Tabellenfunktionen mit mehr als zwei Dimensionen verzichtbar.

Beispiele:

```
TABULAR FUNCTION  Parabel (REAL --> REAL)
CONTINUOUS
BY CUBIC BESSEL INTERPOLATION
ON  (-1.5,  -1.0, -0.5,  0.0, 0.5,  1.0, 1.5)
--> (-2.25, -1.0, -0.25, 0.0, 0.25, 1.0, 2.25)

TABULAR FUNCTION  Not (LOGICAL --> LOGICAL)
ON  (FALSE, TRUE)
--> (TRUE,  FALSE)

TABULAR FUNCTION  And (LOGICAL, LOGICAL --> LOGICAL)
ON (           (FALSE, TRUE);
     FALSE --> (FALSE, FALSE),
     TRUE  --> (FALSE, TRUE)   )
```

Kontinuierliche Tabellenfunktionen lassen sich noch dahingehend verbessern, daß man Unstetigkeitsstellen (Knick- und Sprungstellen) vorsieht oder die Angabe der Ableitung optional ermeoglicht, falls diese bekannt ist.

Beispiel:

```
TABULAR FUNCTION  Concut (REAL --> REAL)
CONTINUOUS
BY CUBIC BESSEL INTERPOLATION
ON  (-1.0, -0.5,   0.0 ] 0.0,     1.0)
--> (-1.0, -0.25,  0.0 | 1.0:2.0, 3.0)
```

Die Tabellenfunktion besitzt im Punkt 0.0 eine Sprungstelle. Rechts davon verläuft eine Gerade, links eine Parabel. Die Angabe der Ableitung im Punkt +0.0 ist bei kubischer Interpolation durch zwei Punkte erforderlich.

Wie die Beispiele zeigen, können Tabellenfunktionen mit beträchtlichem Komfort ausgestattet werden. Auch dieser Gesichtspunkt macht die Aufnahme der Funktionstabellen in die Sprache sinnvoll, weil dadurch eine umfangreiche Analyse von Syntax und Semantik der Datensätze erfolgen kann. Um die Auswertung zur Laufzeit, insbesondere die Interpolation kontinuierlicher Funktionen braucht sich der Anwender dann ebenfalls keine Gedanken zu machen.

## 6.7 Schnittstellen zur Umgebung

Die bisher beschriebene Funktionalität des Sprachkonzepts läßt bezüglich der Formulierung von Modellen keine Wünsche mehr offen. Leider ist aber auch noch zu berücksichtigen, daß der Anwender den Simulator in eine bereits vorhandene Umgebung auf seinem Rechner einbetten möchte.

So möchte er beispielsweise vor oder während eines Simulationslaufs Daten einlesen, die dem Betrieb des Modells dienen. Dabei handelt es sich nicht etwa um Parameter, die ein spezielles Modell charakterisieren, sondern etwa um Auftragsdaten oder Arbeitspläne. Diese Informationen benötigt der Anwender auch noch für andere Zwecke und ist daher an einer zentralen Datenhaltung interessiert. Diese Daten sind nicht selten in Datenbanken oder in einem benutzereigenen Format abgelegt.

Ebenso besteht die Notwendigkeit einer Datenausgabe, die über die Möglichkeiten der angebotenen Beobachter hinausgeht, wenn Ausgaben aufgrund besonderer Bedingungen und in speziellen Formaten erfolgen sollen.

Die Wünsche der Anwender in Bezug auf Ein-/Ausgabe sind sehr vielfältig und schwer genauer zu spezifizieren. Offensichtilch kann man diesen Wünschen nur nachkommen, wenn man die Möglichkeit anbietet, Funktionen der Basissprache (C) aus dem Simulationsprogramm heraus aufzurufen.

Für diese Lösung spricht auch eine weitere Anforderung: Komplexere Steuerstrategien liegen häufig bereits als Programme in einer höheren Programmiersprache vor. Durch die Übertragung in eine andere Sprache würden sich nur Fehler einschleichen und man würde der Möglichkeit beraubt per Simulation das Originalprogramm zu testen. Aus diesen Gründen - und natürlich wegen des Zeitaufwands - ist es erwünscht, das Originalprogramm unverändert mit einzubeziehen.

Da es ohnehin als notwendig erachtet wurde, Funktionen und Prozeduren anzubieten, kann dem auch sehr leicht Rechnung getragen werden. Es ist nämlich lediglich erforderlich offenzulegen, welchen Code der Compiler generiert, wenn er eine Funktion oder Prozedur aufruft und wie dieses Unterprogamm die Parameter entgegennimmt. Der Anwender hat sich nur an diese Konventionen zu halten, damit anstelle einer MDL-Funktion eine C-Funktion durchlaufen wird.

In der Modellbeschreibung steht dann anstelle einer MDL-Funktion die Deklaration einer C-Funktion, damit der Compiler Anzahl und Typen der Parameter des Funktionsaufrufs prüfen kann.

Beispiel:

```
C_FUNCTION  Strategie  (INT, ARRAY[n] REAL --> INT)
```

So einfach dieses Konzept klingen mag, die Schnittstellenbeschreibung erreicht doch einen beträchtlichen Umfang. Insbesondere der Umgang mit Feldern nötigt dem Anwender einigen Einarbeitungsaufwand ab.

## 6.8 Simultanbetrieb mit anderen Prozessen

Beim Einsatz eines Simulationsprogramms als Trainingssimulator wird mit einem offenen Modell gearbeitet, d.h. das Modell bzw. das Simulationsprogramm reagiert fortwährend auf

Eingaben aus der Umgebung. Diese Eingaben geben die Werte von Sensorvariablen vor, die nicht an andere Modellkomponenten angeschlossen sind. Zu diesem Zweck benötigt das Simulationsprogramm eine Schnittstelle, die es anderen Prozessen ermöglicht, diese Informationen zuzusenden.

Die eingehenden Informationen haben etwa folgenden Inhalt:

- Kennung: Setzen einer Sensorvariablen
- Name der Sensorvariablen bzw. deren Adresse im Simulationsprogramm
- Neuer Wert der Sensorvariablen

Diese Vorgehensweise erfordert jedoch gleichzeitig eine Synchronisation mit der Echtzeit, damit der Simulator nicht seiner Umgebung in der Zeit vorauseilt.

Ebenso ist das Abfragen von Variablenwerten von Interesse:

- Kennung: Abfragen eines Variablenwertes
- Name der Variablen bzw. deren Adresse

Die Ausgabe erfolgt dann über einen zweiten Kanal:

- Kennung: Mitteilung eines Variablenwertes
- Name der Variablen bzw. deren Adresse
- Wert der Variablen

Schnittstellen wie die eben beschriebenen machen den Simulator zu einem eigenständigen Modul in einem Verbund von Prozessen. Betriebsbegleitende Simulation wird dadurch ebenso ermöglicht wie das Testen von Steuerungs-Software an einer simulierten Anlage.

# Kapitel 7

# Anwendungen

Die folgenden Anwendungsbeispiele sollen die Tragfähigkeit des vorgestellten Sprachkonzepts unter Beweis stellen und dem Leser einen Eindruck davon vermitteln, welche Möglichkeiten sich hiermit eröffnen.

Es geht dabei mehr darum, den Einsatz der sprachlichen Mittel zu demonstrieren als für jedes Anwendungsgebiet ein typisches Beispiel abzugeben. Das würde den Rahmen der Arbeit sprengen. Der interessierte Leser sei hierzu auf [ApEsWi 92] verwiesen.

## 7.1 Die Modellbank "PetriNet"

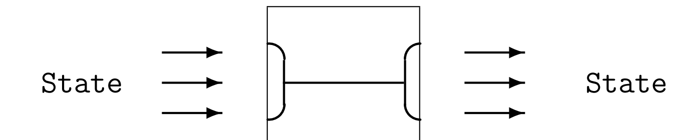
Die Modellbank PetriNet stellt die Komponenten zum Aufbau von allgemeinen Petrinetzen zur Verfügung. Die Marken sind durch Zähler repräsentiert.

Dieses Beispiel zeigt, daß mit wenig Aufwand eine ganze Modellklasse erschlossen werden kann. Die eigentliche Modellierung findet auf der Ebene der höheren Komponenten durch Verschaltung statt. Es ist leicht vorstellbar, daß durch eine zusätzliche graphische Bedienoberfläche das Arbeiten mit dem System wesentlich attraktiver gemacht werden kann.

Komponentenklassen:

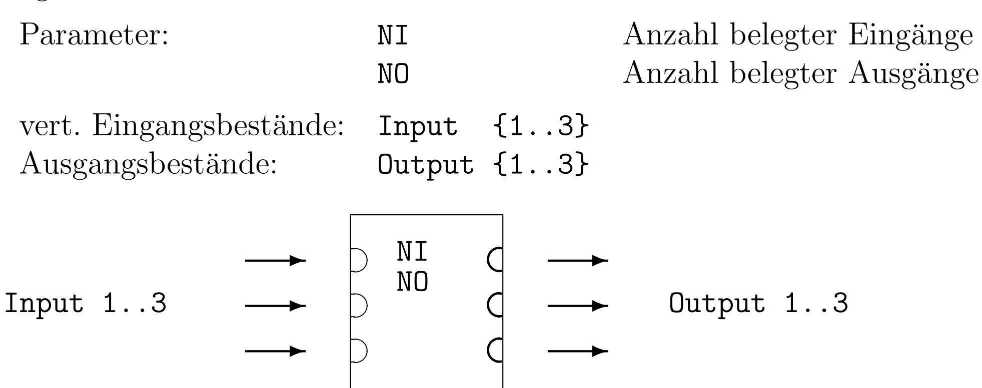
- *Place*: Eine Stelle (Place) ist eine Bestandsvariable, die mit einer, keiner oder mehreren Marken belegt sein kann. Diese Bestandsvariable kann mit einem oder mehreren Eingängen und einem oder mehreren Ausgängen von Transitionen verbunden werden. Die Belegungen der Stellen repräsentieren den Zustand des Petri-Netzes.

  Durchgangsbestand: `State`



- *Transition*: Eine Transition feuert, wenn alle Eingänge durch mindestens eine Marke belegt sind. In diesem Fall wird jeder Stelle auf der Eingangsseite je eine Marke entnommen und jeder Stelle auf der Ausgangsseite je eine Marke zugeführt.

| | | |
|---|---|---|
| Parameter: | `NI` | Anzahl belegter Eingänge |
| | `NO` | Anzahl belegter Ausgänge |
| vert. Eingangsbestände: | `Input {1..3}` | |
| Ausgangsbestände: | `Output {1..3}` | |




Dieses Modell macht folgendes deutlich: Eine Stelle ist als `TRANSIT LOCATION` zu deklarieren, da diesem Platz sowohl Marken hinzugefügt wie auch weggenommen werden. Die Eingänge der Transitionen sind als `DENTRY LOCATIONS` deklariert. Das besagt, daß mehrere Transitionen mit derselben Stelle verbunden sein dürfen. Dadurch ist es möglich, daß zwei oder mehrere Transitionen gleichzeitig je eine Marke von der Stelle abziehen. Darin steckt natürlich eine Konfliktmöglichkeit, weil u.U. mehr Marken von einer Stelle weggenommen werden sollen, als sich auf ihr befinden.

```
# ------------------------------------------------------------------------------

BASIC COMPONENT  Place

DECLARATION

   TRANSIT LOCATION  State (INT) :=  0

END OF  Place

# ------------------------------------------------------------------------------

BASIC COMPONENT  Transition

DECLARATION
   STATE VARIABLES
   CONSTANT         NI (INT) := 1 ,
                    NO (INT) := 1

   DENTRY LOCATION  ARRAY {1..3} Input  (INT) := 1

   EXIT   LOCATION  ARRAY {1..3} Output (INT)

   SOURCE LOCATION  Source (INT),
   SINK   LOCATION  Sink   (INT)
```

```
DYNAMIC BEHAVIOUR

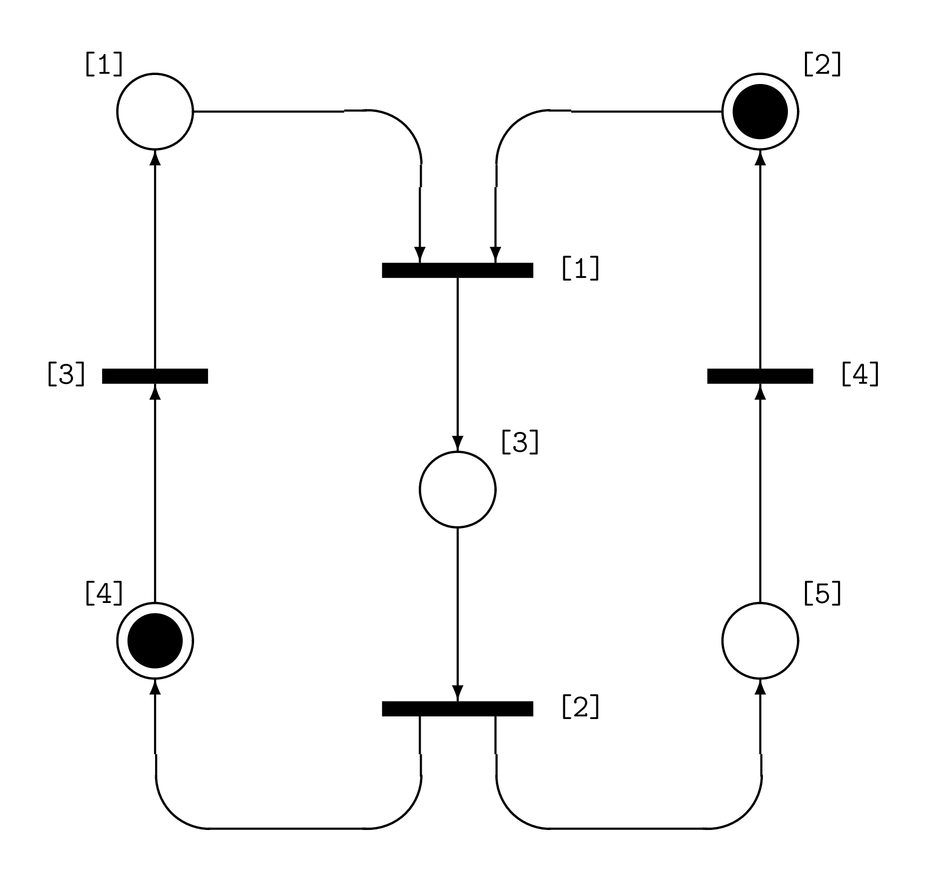
   IF  ALL { i IN 1..3 |  Input[i] > 0 }
   LET
       FOREACH  i IN 1..3  |  i <= NI
       LET
           (Input [i] -> Sink) :=  1;
       END

       FOREACH  i IN 1..3  |  i <= NO
       LET
           (Source -> Output [i]) :=  1;
       END
   END

END OF  Transition

# ----------------------------------------------------------------------------
```

Aus diesen beiden Grundbausteinen lassen sich nun ganze Netze wie etwa das folgende aufbauen:



```
HIGH LEVEL COMPONENT  System

SUBCOMPONENTS   ARRAY {1..5}  Place ,
                ARRAY {1..4}  Transition

STRUCTURE
   TRANSPORT CONNECTIONS

   Place [1]. State  ==>  Transition [1]. Input [1];
   Place [2]. State  ==>  Transition [1]. Input [2];

   Transition [1]. Output [1]  ==>  Place [3]. State;

   Place [3]. State  ==>  Transition [2]. Input [1];

   Transition [2]. Output [1]  ==>  Place [4]. State;
   Transition [2]. Output [2]  ==>  Place [5]. State;

   Place [4]. State  ==>  Transition [3]. Input [1];
   Place [5]. State  ==>  Transition [4]. Input [1];

   Transition [3]. Output [1]  ==>  Place [1]. State;
   Transition [4]. Output [1]  ==>  Place [2]. State;

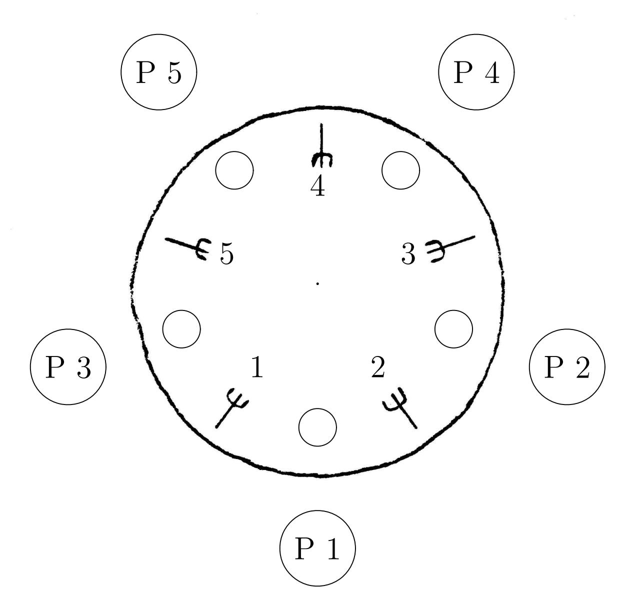
INITIAL CONDITIONS
   Place [2]. State :=  1;
   Place [4]. State :=  1;

   Transition [1]. NI :=  2;
   Transition [2]. NO :=  2;

END OF  System
```

## 7.2 Das Fünf-Philosophen-Problem

Als Standardbeispiel zur Demonstration von Verklemmungen hat sich das fünf-Philosphen-Problem bewährt. Es wird angenommen, daß dieses Problem dem Leser bekannt ist.




Die Gabeln sind dabei Elemente, deren Anzahl stets konstant bleibt. Eine Gabel kann sich auf dem Tisch, in der linken Hand des rechts sitzenden Philosphen oder in rechten Hand des links sitzenden Philosphen befinden.

Dieses Modell macht folgendes deutlich: Der Platz auf dem Tisch, an dem sich die Gabel befindet, ist als `TRANSIT LOCATION` zu deklarieren, da die Gabel sowohl von diesem Platz weggenommen als auch hingebracht wird. Da auch jeder Philosoph die Gabeln in beiden Richtungen bewegt, greift auch er über eine `TRANSIT LOCATION` zu. Den Platz, an dem eine Gabel liegt, teilen sich somit drei Komponenten. Alle drei verfügen über die gleichen Rechte, auf die Gabeln zuzugreifen.

Aus der erforderlichen Deklaration wird dem Anwender bewußt, daß es für diesen Platz einen Zugriffskonflikt gibt und sich je zwei Philosophen darauf verständigen müssen, wer die Gabel erhält, wenn beide sie gleichzeitig wollen. Daß dieser Fall im nachfolgenden Modell nicht berücksichtigt ist, zeigt allerdings auch, daß es bei modularem Modellaufbau möglich ist, semantisch unvollständige Modelle zu formulieren, ohne daß dies vom Compiler verhindert werden könnte.

Läge während des Simulationslaufs tatsächlich ein gleichzeitiger Zugriff vor, dann würde der Zähler auf dem Platz des Tisches eine -1 annehmen. Es ist Sache des Laufzeitsystems, in einem solchen Fall den Simulationslauf mit einer Fehlermeldung zu beenden. Je grober das zeitliche Auflösungsvermögen eingestellt ist, desto häufiger kann eine solche Situation eintreten.

In erster Linie wird das Modell jedoch als Beispiel für ein anderes Problem herangezogen. Wenn sich alle Philosophen etwa zur gleichen Zeit dem Essen zuwenden und ihre linke Gabel aufnehmen, kann keiner von ihnen mehr die rechte Gabel nehmen und zu essen beginnen. Das System hat sich verklemmt, da keine weitere Zustandsänderung mehr möglich ist. Damit

dieser Fall eintreten kann, ist es notwendig, daß jeder Philosoph eine gewisse Zeit braucht, um seine linke Gabel aufzunehmen, und erst danach beginnt, seine rechte Gabel aufzunehmen.

Die Möglichkeit eines solchen Verhaltens ist der Modellspezifikation leider nicht zu entnehmen. Das Eintreten einer Verklemmung zeigt erst ein genügend langer Simulationslauf. Das Laufzeitsystem ist in der Lage, Situationen zu erkennen, in denen keine weiteren Aktivitäten mehr möglich sind. Der Anwender kann einen solchen Zustand dann ausgiebig analysieren.

Die Überprüfung einer Modellspezifikation auf korrekte Semantik besagt lediglich, daß es möglich ist, nach der Spezifikation ein reales System zu bauen, ohne daß Informationen fehlen oder sich widersprechen. Deadlocks sind aber ebenso wie Lifelocks Phänomene, die sowohl im Modell wie auch im realen System in gleicher Weise beobachtbar sind.

Wenn hingegen der Zugriff nicht eindeutig geregelt ist, liegt eine etwas andere Situation vor. Ein reales System, das aus Komponenten aufgebaut ist, die keine Vereinbarung über einen Zugriffskonflikt getroffen haben, werden ein Verhalten zeigen, das nicht im voraus geregelt war.

Ein Simulationsprogramm, das aus einer (unvollständigen) Modellbeschreibung hervorgeht, weiß natürlich nicht, wie sich das reale System in einer solchen Situation verhält. Der Modellablauf kann daher nur mit einer Fehlermeldung beendet werden.

Zur Vermeidung dieser Problematik könnte man das System im Sinne der objektorientierten Programmierung modellieren. Die Plätze wären Komponenten (Objekte), die auf Anfrage selbst entscheiden, ob sie dem rechten oder linken Philosophen die Gabel aushändigen. Das Objekt Platz entscheidet sich in jedem Fall eindeutig.

Im Gegensatz zur objektorientierten Programmierung ist es aber nicht unser Anliegen, ein Sprachkonzept zu entwerfen, das nur softwaretechnisch sichere Lösungen zuläßt. Die Modellbeschreibungssprache will vielmehr vorgefundene Gegebenheiten nachbilden und die damit verbundenen Probleme deutlich werden lassen.

```
# -----------------------------------------------------------------------------

BASIC COMPONENT  Philosopher

DEFINITIONS
   VALUE SET  PhilStates: ('Thinking', 'GetLeft', 'GetRight', 'Eating')

DECLARATIONS
   STATE VARIABLES   State (PhilStates) := 'Thinking' ,  # aktueller Zustand
                     TActEnd (REAL)     :=  0 ,          # Ende einer Aktivitaet

   RANDOM VARIABLES  TGetFork (REAL):  UNIFORM (LowLimit := 0.1, UpLimit := 1) ,
                     TEat     (REAL):  EXPO (Mean := 100) ,
                     TThink   (REAL):  EXPO (Mean := 500)

   PRIVATE LOCATION  LeftHand  (INT) := 0 ,
                     RightHand (INT) := 0

   TRANSIT LOCATION  LeftSide  (INT) := 0 ,
                     RightSide (INT) := 0
```

```
DYNAMIC BEHAVIOUR

   IF  State = 'Thinking'  AND  T >= TActEnd       # Aufnehmen der linken Gabel
   LET
       IF  CARD LeftSide > 0
       LET
           State^   := 'GetLeft';
           TActEnd^ :=  T + TGetFork;

           (LeftSide -> LeftHand) := 1;
       END
   END
   ELSIF  State = 'GetLeft'  AND  T >= TActEnd     # Aufnehmen der rechten Gabel
   LET
       IF  CARD RightSide > 0
       LET
           State^   := 'GetRight';
           TActEnd^ :=  T + TGetFork;

           (RightSide -> RightHand) := 1;
       END
   END
   ELSIF  State = 'GetRight'  AND  T >= TActEnd    # Essen
   LET
       State^   := 'Eating';
       TActEnd^ :=  T + TEat;
   END
   ELSIF  State = 'Eating'  AND  T >= TActEnd      # Ablegen der Gabeln
   LET
       State^   := 'Thinking';
       TActEnd^ :=  T + TThink;

       (RightHand -> RightSide) := 1;
       (LeftHand  -> LeftSide)  := 1;
   END

END OF  Philosopher

# ------------------------------------------------------------------------------

BASIC COMPONENT  Table

DECLARATION
  TRANSIT LOCATION  ARRAY {1..5} Place (INT) := 1

END OF  Table

# ------------------------------------------------------------------------------
```

```
HIGH LEVEL COMPONENT  System

SUBCOMPONENTS  ARRAY {1..5} Philosopher, Table

STRUCTURE
   TRANSPORT CONNECTIONS
   Philosopher[1].RightSide <==> Table.Place[2] <==> Philosopher[2].LeftSide;
   Philosopher[2].RightSide <==> Table.Place[3] <==> Philosopher[3].LeftSide;
   Philosopher[3].RightSide <==> Table.Place[4] <==> Philosopher[4].LeftSide;
   Philosopher[4].RightSide <==> Table.Place[5] <==> Philosopher[5].LeftSide;
   Philosopher[5].RightSide <==> Table.Place[1] <==> Philosopher[1].LeftSide;

INITIAL CONDITIONS
   Philospher[1]. TActEnd :=  1;       # Unterschiedliches Ende der
   Philospher[2]. TActEnd :=  2;       # Anfangsaktivitaet
   Philospher[3]. TActEnd :=  3;
   Philospher[4]. TActEnd :=  4;
   Philospher[5]. TActEnd :=  5;

END OF  System
```

## 7.3 Das Modell “Bodenfeuchte”

Das Modell Bodenfeuchte ist ein Beispiel für die Modellierung mit Bestandsgrößen bei kontinuierlichen Vorgängen. Es beschreibt den Wasserhaushalt in einem Wald [Boss 85, May89].

Regen, der auf einen Wald fällt, landet zunächst auf dem Laub der Bäume und fällt erst dann zur Erde, wenn sich auf dem Laub bereits eine gewisse Menge an Wasser gesammelt hat. Das Wasser auf dem Laub verdunstet nach und nach wieder (Interzeption).

Das Wasser, das auf den Boden fällt, dringt relativ rasch in die oberste Bodenschicht (AH-Horizont) ein. Das Wasser in dieser humosen Bodenschicht wird zum Teil von den Wurzeln der Bäume aufgenommen, zum Teil verdunstet es wieder (Evaporation) und zum Teil versickert es in die nächst tiefere Bodenschicht (B-Horizont).

Das Wasser im B-Horizont ist für die Wurzeln nicht mehr erreichbar. Allerdings findet ein kapillarer Aufstieg in den AH-Horizont statt, wenn der Boden dort trockener ist als im B-Horizont. Wasser, das noch tiefer versickert, wird im Modell nicht mehr berücksichtigt.

Das Wasser, das von den Wurzeln der Bäume aufgenommen wird, verdunstet nach und nach über die Blätter (Transpiration).

Das Modell zeigt, daß durch die transportbezogene Beschreibung mit Bestandsgrößen

a) eine dem Modell äquivalente graphische Darstellung (System Dynamics) möglich ist und

b) eine formale Modellbeschreibung entsteht, die einer informellen, natürlich-sprachlichen Beschreibung unmittelbar entspricht.

Darüber hinaus zeigt es, daß auch eine System Dynamics Darstellung auf einfache Art und Weise in Module zerlegt werden kann. Für ein Zusammenwirken der beiden Komponenten `Wasser` und `Baum` ist lediglich der Bestand `FHumus` als TRANSIT LOCATION bzw. DENTRY LOCATION zu deklarieren.

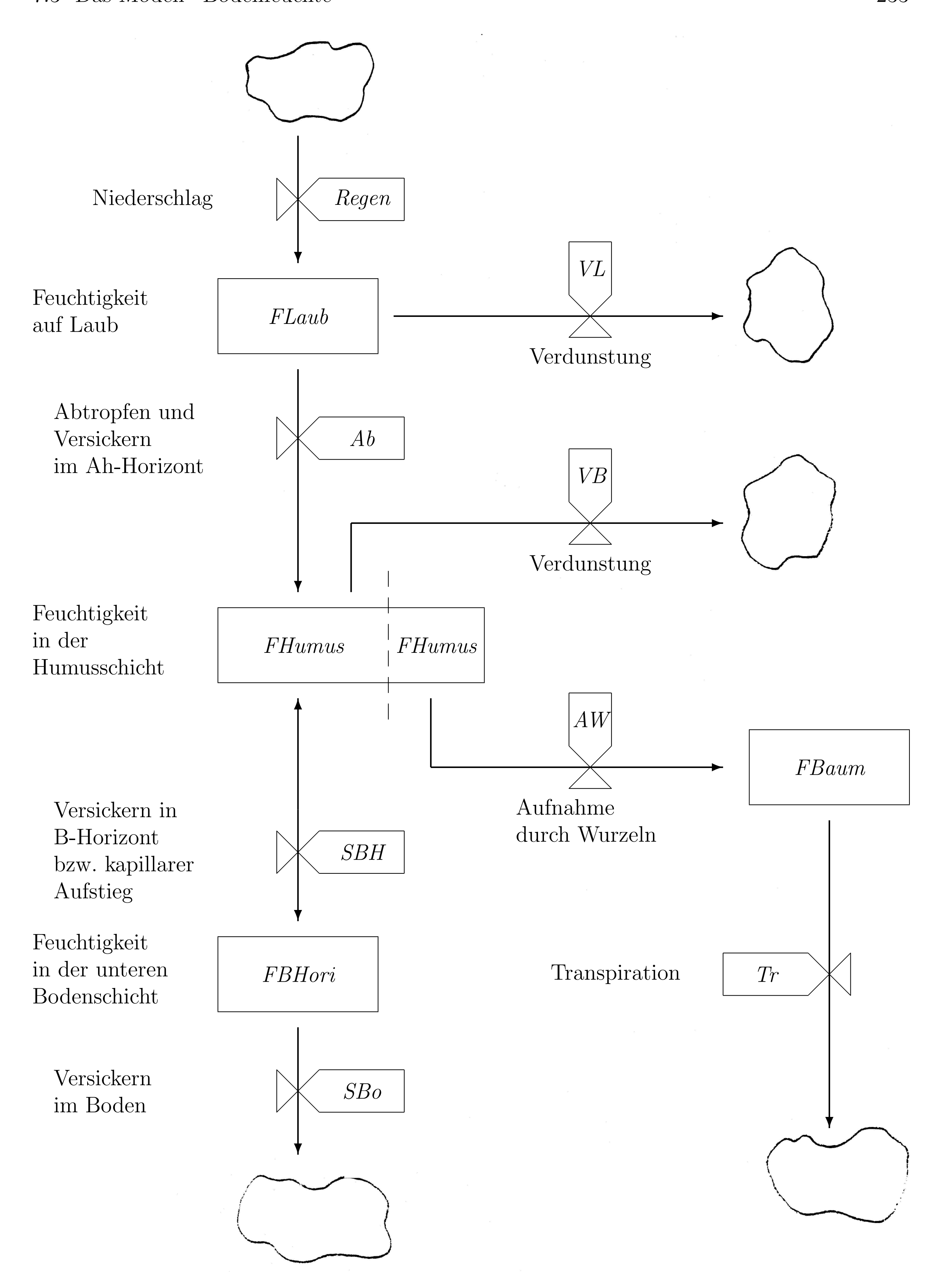



Abb. 7.3-1: Modell "Bodenfeuchte" in System Dynamics Darstellung

```
BASIC COMPONENT  Wasser

DECLARATIONS
   PRIVATE LOCATIONS  FLaub  (REAL) :=  0 ,
                      FBHori (REAL) := ...

   TRANSIT LOCATION   FHumus (REAL) := ...

   SOURCE  LOCATION   Wolken (REAL)

   SINK    LOCATION   Atmosphaere (REAL), Grundwasser (REAL)


DYNAMIC BEHAVIOUR

   (Wolken -> FLaub)'       :=  ... ;      #  Niederschlagsmenge pro Zeit;

   (FLaub  -> Atmosphaere)' :=  ... ;      #  Verdunstung auf Blaettern
                                           #  (Interzeption)

   (FLaub  -> FHumus)'      :=  ... ;      #  Abtropfen und Versickern
                                           #  in Humusschicht

   (FHumus -> Atmosphaere)' :=  ... ;      #  Bodenverdunstung

   (FHumus -> FBHori)'      :=  ... ;      #  Versickern in B-Horizont
                                           #  bzw. kapillarer Aufstieg in Ah

   (FBHori -> Grundwasser)' :=  ... ;      #  Versickern in tieferen Schichten

END OF  Wasser

# ------------------------------------------------------------------------------

BASIC COMPONENT  Baum

DECLARATION

   PRIVATE LOCATION  FBaum  (REAL) :=  ...
   DENTRY  LOCATION  FHumus (REAL)              #  Verzicht auf Initialisierung
   SINK    LOCATION  Atmosphaere (REAL)

DYNAMIC BEHAVIOUR

   (FHumus -> FBaum)'       :=  ... ;         #  Wasseraufnahme durch Wurzeln

   (FBaum  -> Atmosphaere)' :=  ... ;         #  Verdunstung durch Baum
                                              #  (Transpiration)

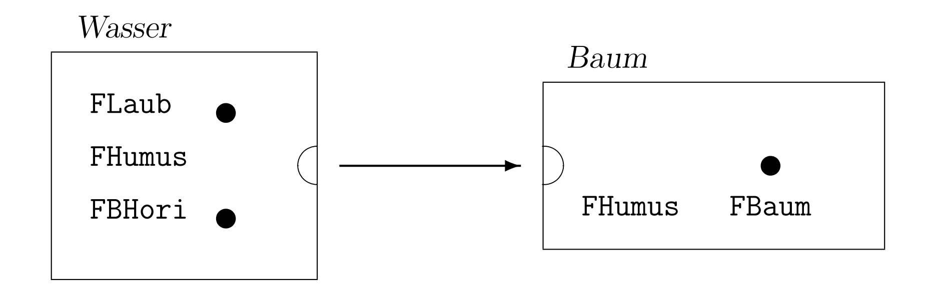
END OF  Baum
```

```
HIGH LEVEL COMPONENT  Bodenfeuchte

SUBCOMPONENTS  Wasser, Baum

STRUCTURE
   TRANSPORT CONNECTION
   Wasser.FHumus  ==>  Baum.FHumus;

END OF  Bodenfeuchte
```



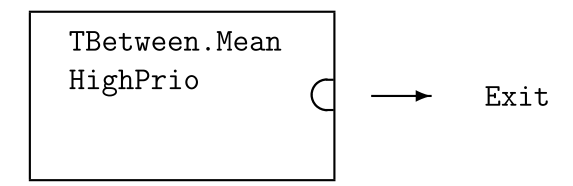

## 7.4 Die Modellbank “QueueNet”

Die Modellbank QueueNet stellt Komponenten zum Aufbau von einfachen Warteschlangenmodellen zur Verfügung wie sie in der Warteschlangentheorie behandelt werden. Als Attribute führen die mobilen Elemente eine Priorität mit und den Zeitpunkt, zu dem ihre Bearbeitung endet. Der Wert der Priorität kann 0 (niedrige Priorität) oder 1 (hohe Priorität) betragen.

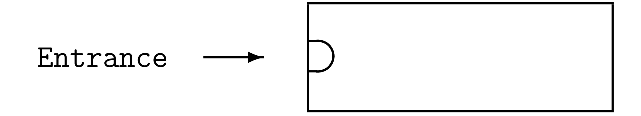
Die Modellbank enthält die folgenden Komponenten:

- *Job*: Auftrag, der das Modell durchläuft
- *Source*: Quelle, die mobile Elemente mit einer exponentiell verteilten Zwischenankunftszeit erzeugt

  | | | |
  |---|---|---|
  | Parameter: | `TBetween.Mean` | mittlere Zwischenankunftszeit |
  | | `HighPrio` | Wahrscheinlichkeit für hohe Priorität |
  | Ausgang: | `Exit` | |



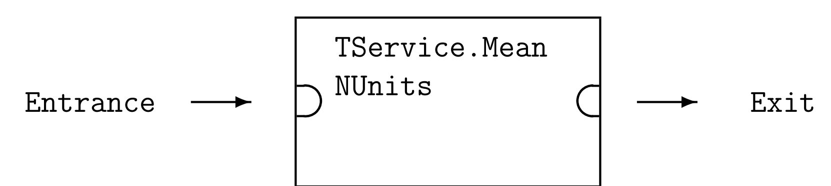

- *Senke*: Senke, die mobile Elemente vernichtet



- *Facility*: Warteschlange mit einer oder mehreren Bedieneinheiten und exponentialverteilter Bedienzeit

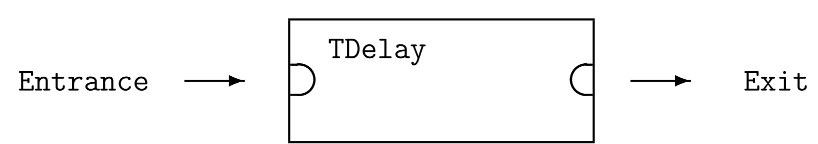
| | | |
|---|---|---|
| Parameter: | `TService.Mean` | mittlere Bedienzeit |
| | `NUnits` | Anzahl von Bedieneinheiten |

Eingang: `Entrance`
Ausgang: `Exit`




- *Advance*: Zeitverzögerung eines oder mehrerer Aufträge um eine feste Zeit

Parameter: `TDelay` Verzögerungszeit
Eingang: `Entrance`
Ausgang: `Exit`




- *Distrib*: Verteilung von Aufträgen nach Wahrscheinlichkeiten auf drei verschiedene Ausgänge

| | | |
|---|---|---|
| Parameter: | `p1` | Wahrscheinlichkeit für Exit1 |
| | `p2` | Wahrscheinlichkeit für Exit2 |

Eingang: `Entrance`

Ausgänge: `Exit1`
`Exit2`
`Exit3`

p1
p2
Entrance
Exit1
Exit2
Exit3

Nach Ablauf der Bearbeitungszeit schickt die Bedienstation den Auftrag weiter. Deshalb sind alle Komponenten so ausgelegt, daß sie jederzeit einen Auftrag entgegennehmen können. Insbesondere ist es erforderlich, daß vor einer Bedieneinheit stets eine Warteschlange angeordnet ist.

Alle Transporte sind gerichtet. Eingänge dürfen nicht parallel zueinander liegen, für Ausgänge ist dies erlaubt. Das bedeutet, daß Transporte zwischen Komponenten zum Sammeln von Aufträgen dienen dürfen, aber es darf keine Verteilung auf mehrere Eingänge stattfinden. Aus diesem Grund muß zum Verteilen eine eigene Komponente existieren.

Mit den Komponenten dieser Modellbank ist der Sprachumfang der Simulationsssprache GPSS bereits nahezu abgedeckt.

```
MOBILE COMPONENT  Job

DECLARATIONS
   STATE VARIABLES
   CONSTANT        Priority (INT)  :=  0       # Prioritaet
   DISCRETE        TReady   (REAL) :=  0       # Fertigstellungszeitpunkt

END OF  Job

# ------------------------------------------------------------------------------

BASIC COMPONENT  Source

MOBILE SUBCOMPONENTS OF CLASS  Job

DECLARATIONS
   STATE VARIABLE
   CONSTANT          HighPrio (REAL) :=  0          # Wahrscheinlichkeit hoher
                                                    # Prioritaet
   DISCRETE          TNext    (REAL) :=  0          # Zeitpunkt der naechsten
                                                    # Generierung

   RANDOM VARIABLES  Random   (REAL) :  UNIFORM (LowLimit := 0, UpLimit := 1) ,
                     TBetween (REAL) :  EXPO (Mean := 10)

   EXIT   LOCATION   Exit (SET OF Job) :=  0 Job

   SOURCE LOCATION   JobSource (SET OF Job)

DYNAMIC BEHAVIOUR

   IF  T >=  TNext
   LET
       TNext^ :=  T + TBetween;

       FOREACH  J  IN  JobSource: Job{1}
       LET
           J. Priorty^ :=  1  WHEN  Random <= HighPrio  ELSE
                           0  END
           J -> Exit;
       END
   END

END OF  Source
```

```
BASIC COMPONENT  Sink

MOBILE SUBCOMPONENTS OF CLASS  Job

DECLARATION
   ENTRY LOCATION   Entrance (SET OF Job)
   SINK  LOCATION   JobSink  (SET OF Job)

DYNAMIC BEHAVIOUR

   FOREACH  J  IN  Entrance
   LET
       J -> JobSink;
   END

END OF  SinkJob

# ------------------------------------------------------------------------------

BASIC COMPONENT  Facility

MOBILE SUBCOMPONENTS OF CLASS  Job

DECLARATIONS
   STATE VARIABLE
   CONSTANT         NUnits   (INT)  :=  1        # Anzahl der Bedieneinheiten

   RANDOM VARIABLE   TService (REAL) :  EXPO (Mean := 8)

   PRIVATE LOCATION  Server (SET OF Job  ||  INC TReady) :=  0 Job

   ENTRY   LOCATION  Queue  (SET OF Job  ||  DEC Priority)
   EXIT    LOCATION  Exit   (SET OF Job)

DYNAMIC BEHAVIOUR

#  Einlasten in eine der Bedieneinheiten
#  -------------------------------------
   IF  CARD Server < NUnits
   LET
       FOREACH  J  IN  Queue:Job{1}
       LET
           J -> Server;

           J.TReady^ :=  T + TService;
       END
   END
```

```
#  Weiterleiten nach Fertigstellung
#  --------------------------------
   FOREACH  J  IN  Server:Job{1}
   LET
       IF  T >= J.TReady
       LET
           J -> Exit;
       END
   END

END OF  Facility

# ------------------------------------------------------------------------------

BASIC COMPONENT  Advance

MOBILE SUBCOMPONENTS OF CLASS  Job

DECLARATIONS
   STATE VARIABLE
   CONSTANT          TDelay (REAL) :=  8

   PRIVATE LOCATION  Wait (SET OF Job) :=  0 Job

   ENTRY  LOCATION   Entrance (SET OF Job)
   EXIT   LOCATION   Exit     (SET OF Job)

DYNAMIC BEHAVIOUR

#  Ankommender Auftrag
#  -------------------
   FOREACH  J  IN  Entrance:Job{1}
   LET
       J -> Wait;

       J.TReady^ :=  T + TDelay;
   END

#  Weiterleiten nach Ablauf der Verzoegerungszeit
#  ----------------------------------------------
   FOREACH  J  IN  Wait:Job{1}
   LET
       IF  T >= J.TReady
       LET
           J -> Exit;
       END
   END

END OF  Advance
```

```
BASIC COMPONENT  Distrib

MOBILE SUBCOMPONENTS OF CLASS  Job

DECLARATIONS
   STATE VARIABLES
   CONSTANT          p1    (REAL) :=  0.5 ,
                     p2    (REAL) :=  0.5

   RANDOM VARIABLES  Random (REAL) :  UNIFORM (LowLimit := 0, UpLimit := 1)

   ENTRY LOCATION    Entrance (SET OF Job)

   EXIT   LOCATION   Exit1 (SET OF Job) ,
                     Exit2 (SET OF Job) ,
                     Exit3 (SET OF Job)

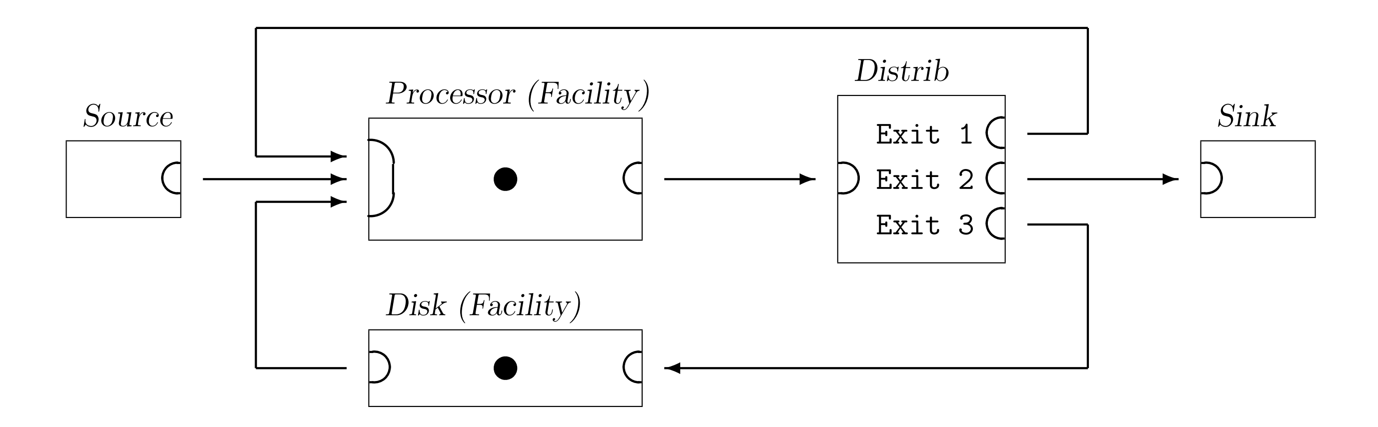
DYNAMIC BEHAVIOUR

   FOREACH  J  IN  Entrance:Job{1}
   LET
       IF     Random <= p1       LET  J -> Exit1;  END
       ELSIF  Random <= p1 + p2  LET  J -> Exit2;  END
       ELSE                      LET  J -> Exit3;  END
   END

END OF  Distrib
```

## Modell einer einfachen Rechenanlage

Durch Zusammenschalten der gezeigten Komponenten lassen sich im Grunde alle Modelle der Warteschlangentheorie untersuchen. Das folgende Beispiel zeigt eine stark vereinfachte Rechenanlage als offenes Warteschlangenmodell.



```
HIGH LEVEL COMPONENT  Computer

SUBCOMPONENTS  Source,
               Processor OF CLASS Facility,
               Disk      OF CLASS Facility,
               Distrib,
               Sink

STRUCTURE
   TRANSPORT CONNECTIONS

   (Source.Exit == Disk.Exit == Distrib.Exit1)  ==>  Processor.Entrance;

   Processor.Exit  ==>  Distrib.Entrance;
   Distrib.Exit2   ==>  Disk.Entrance;
   Distrib.Exit3   ==>  Sink.Entrance;

END OF  Computer
```

Will man die Quelle und Senke nach außen verlagern und die höhere Komponente mit entsprechenden Schnittstellen ausstatten, erhält man folgende Modellkomponente:

```
HIGH LEVEL COMPONENT  Computer

SUBCOMPONENTS  Processor OF CLASS Facility,
               Disk      OF CLASS Facility,
               Distrib

DECLARATIONS
   TRANSIT VARIABLE
   Entrance  <==>  (Disk.Exit == Distrib.Exit1)  ==>  Processor.Entrance;

   EXIT VARIABLE
   Exit  <==  Distrib.Exit3;


STRUCTURE
   TRANSPORT CONNECTIONS
   Processor.Exit  ==>  Distrib.Entrance;
   Distrib.Exit2   ==>  Disk.Entrance;

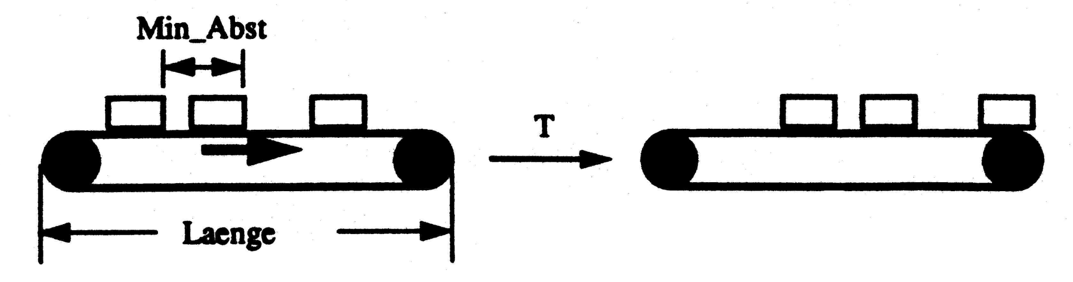
END OF  Computer
```

Da auf die Eingangsbestand des Prozessors auch mobile Elemente gebracht werden, darf die Location 'Entrance' der höheren Komponente leider nur als Durchgangsbestand und nicht als Eingangsbestand deklariert werden.

## 7.5 Die Modellkomponente Transportband

Die Modellkomponente Transportband ist Teil der Modellbank Fertig [ApEsWi 92], in der Komponenten zur Modellierung von Fertigungslinien zusammengestellt sind.

Die Werkstücke sind mobile Elemente, welche die Anlage durchlaufen. Neben der gezeigten Komponente enthält diese Bank Komponenten für Quellen und Senken, Bearbeitungsstationen, Prüfstationen, Bänder als Pufferstrecken oder Sammelstrecken sowie Verteiler und Zusammenführungen.




Die Werkstücke (Teile) werden prinzipiell von der nachfolgenden Komponente geholt. Es gilt die Vereinbarung, daß ein Teil immer dann geholt werden darf, wenn es auf einem Übergabeplatz liegt. Durch das Holen der Teile (im Gegensatz zum Bringen) gelangen nie mehr Werkstücke in eine Komponente als darin Platz finden.

Die Komponente Transportband ist ein Beispiel dafür, daß Vorgänge in Warteschlangenmodellen kombiniert diskret/kontinuierlich beschrieben werden können. In Fällen wie diesem führt dies zu einer deutlich einfacheren Modellbeschreibung.

Beschreibung der Modellkomponente *Transportband:*

Das Band bleibt stehen, wenn das vorderste Teil das Ende des Bandes erreicht hat und nicht vom Band genommen wird. Der Abstand der Teile auf dem Band bleibt daher erhalten. Da die Teile eine gewisse Größe besitzen, halten sie auf dem Band einen Mindestabstand ein.

| | | |
|---|---|---|
| Parameter: | `Laenge:` | Länge des Transportbands $[m]$ |
| | `V:` | Fördergeschwindigkeit $[m/s]$ |
| | `MinAbst:` | Mindestabstand zwischen zwei Teilen $[m]$ |
| Eingangsbestand: | `Eingang` | |
| Ausgangsbestand: | `Ausgang` | |

*Transportband*

```
Eingang  ──►  ┌ Laenge   ┐  ──►  Ausgang
              │ V        │
              └ MinAbst  ┘
```

```
MOBILE COMPONENT  Teil

DECLARATIONS
   STATE VARIABLE
   DISCRETE   XEnde (REAL) := 0           #  Ort, an dem das Teil das Ende
                                          #  des Bandes erreicht hat

END OF  Teil
```

```
BASIC COMPONENT  Transportband

MOBILE SUBCOMPONENTS OF CLASS  Teil

DECLARATIONS
   STATE VARIABLES
   CONSTANT    V       (REAL) :=  5   ,    #  Geschwindigkeit des Bandes
               Laenge  (REAL) :=  5   ,    #  Laenge des Bandes
               MinAbst (REAL) :=  0.5      #  Abstand der Teile auf dem Band
   DISCRETE    XLast   (REAL) := -0.5      #  Ort, an dem das letzte Teil eintraf
                                           #  Vorbesetzung: -MinAbst
   CONTINUOUS X        (REAL) :=    0      #  Ort

   DEPENDENT VARIABLE
   DISCRETE    Fahren (LOGICAL)            #  Anzeige, ob Band faehrt

   ENTRY   LOCATION  Eingang (SET OF Teil) := 0  Teil
   PRIVATE LOCATION  Band    (SET OF Teil) := 0  Teil
   EXIT    LOCATION  Ausgang (SET OF Teil) := 0  Teil

DYNAMIC BEHAVIOUR

   Fahren :=  EMPTY Ausgang;               #  Band faehrt, wenn nicht am
                                           #  Ende ein Teil zum Abholen liegt

   IF  Fahren  LET  X' :=  V;  END
         ELSE  LET  X' :=  0;  END

   IF  NOT EMPTY Eingang  AND  X - XLast >= MinAbst   #  Vom Eingang darf ein
   LET                                                #  Teil geholt werden,
       Eingang: Teil{1} -> Band;                      #  wenn der Abstand gross
       Eingang: Teil{1}. XEnde^ :=  X + Laenge;       #  genug ist
       XLast^  :=  X;
   END

   IF  NOT EMPTY Band
   LET
       IF  X >= Band: Teil[1]. XEnde       #  Sobald das Teil am Ende angekommen
       LET                                 #  ist, wird es auf den Uebergabeplatz
           Band: Teil{1} -> Ausgang;       #  gelegt.
       END
   END

END OF  Transportband
```

## 7.6 Das Modell Linienbus

Mobile Komponenten sind nur Datenträger und besitzen keine eigene Dynamik. Da es sich anbietet, Transportmittel als mobile Komponenten zu repräsentieren, wird oft vermutet, daß diese daher kein eigenständiges Verhalten zeigen dürfen.

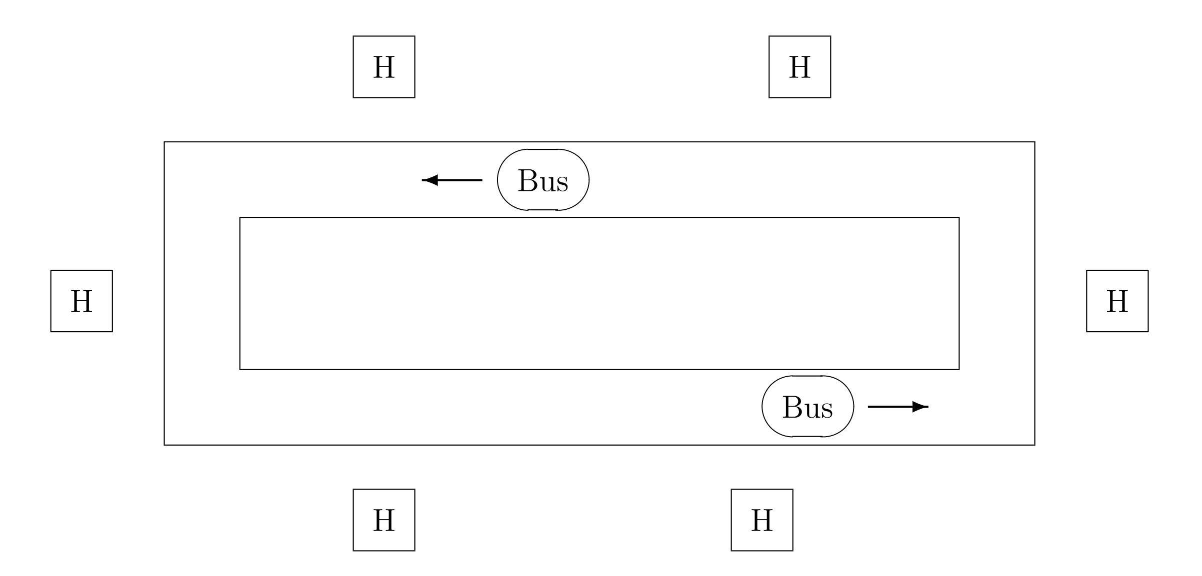
Das folgende Beispiel, ein Linienbus-Verkehr, ist kein auftragsgebundenes Transportsystem, d.h. ein Bus fährt auch dann, wenn gar kein Fahrauftrag vorliegt. Beim auftragsgebundenen Betrieb teilt der Fahrgast seinen Transportwunsch einer Vermittlungsstelle mit. Diese wählt ein Fahrzeug aus und teilt ihm den Transportauftrag mit. Das Abfahren der Strecke jedoch wird in ähnlicher Weise modelliert wie beim Linienbus.

Daher läßt das Beispiel das Prinzip erkennen, nach dem sich auch alle anderen Tranportsysteme wie Taxibetriebe, Aufzüge, fahrerlose Transportsysteme (FTS), Elektrohängebahnen etc. modellieren lassen. Im allgemeinen geht man so vor, daß man alle mobilen Komponenten (Transportmittel) in einer Basiskomponente zusammenfaßt und darin das Abfahren der Strecken beschreibt.

Dieses Modell zeigt weiterhin, wie auf die Location einer mobilen Komponente andere mobile Elemente aufgeladen und auch wieder entladen werden können.

Beschreibung des Modells:

Das Modell Linienbus beschreibt Busse, die auf einem Ringkurs fahren und an den Haltestellen Fahrgäste ein- und aussteigen lassen. An jeder Haltestelle befindet sich eine Quelle, welche Fahrgäste mit ihrem Fahrziel erzeugt. Aussteigende Fahrgäste werden nach ihrer Ankunft in einer Senke vernichtet.




Der Übersichtlichkeit halber ist die Dynamik sehr einfach gehalten. Der Leser hat aber sicherlich keine Schwierigkeiten sich vorzustellen, daß ebenso variable Fahrzeiten, Fahrtunterbrechungen, Einstiegs- und Ausstiegszeiten etc. berücksichtigt werden könnten.

Es wäre auch auf sehr einfache Art möglich, weitere Busse in den Fahrkurs einzuschleusen oder Busse dem Betrieb zu entziehen.

```
MOBILE COMPONENT  Passagier
```

```
DECLARATION
   STATE VARIABLE
   DISCRETE        Ort  (INT) := 1 ,
                   Ziel (INT) := 2

END OF  Passagier

# ------------------------------------------------------------------------------

MOBILE COMPONENT  Bus

MOBILE SUBCOMPONENTS OF CLASS  Passagier

DECLARATION
   STATE VARIABLE  TStation (REAL) := 0 ,    # Ankunft an der naechsten Station
                   StatNo   (INT)  := 0      # aktuelle bzw. Ziel-Station

   PRIVATE LOCATION  Plaetze (SET OF Passagier) := 0 Passagier

END OF  Bus

# ------------------------------------------------------------------------------

BASIC COMPONENT  Quelle

MOBILE SUBCOMPONENTS OF CLASS  Passagier

DEFINITIONS
   TABULAR DISTRIBUTION  RanZiel (INT --> REAL)
   BY LINEAR INTERPOLATION
   ON  (   1,    2,    3, ... ,   10)
   --> (0.15, 0.25, 0.30, ... , 1.00)

DECLARATIONS
   STATE VARIABLES
   DISCRETE          TAnkunft (REAL) := 0

   RANDOM VARIABLES  TZwischen (REAL): EXPO (Mean := 1)
                     RanZiel   (INT):  DistZiel

   EXIT   LOCATION   Ausgang    (SET OF Passagier)

   SOURCE LOCATION   PassQuelle (SET OF Passagier)
```

```
DYNAMIC BEHAVIOUR

   IF  T >= TAnkunft
   LET
       TAnkunft^ :=  T + TZwischen;

       FOREACH  P  IN  PassQuelle: Passagier{1}
       LET
           P. Ziel^ :=  RanZiel;

           P -> Ausgang;
       END

END OF Quelle

# ------------------------------------------------------------------------------

BASIC COMPONENT  Senke

MOBILE SUBCOMPONENTS OF CLASS  Passagier

DECLARATIONS
   ENTRY LOCATION  Eingang   (SET OF Passagier)
   SINK  LOCATION  PassSenke (SET OF Passagier)

DYNAMIC BEHAVIOUR

   Eingang: Passagier {i}  ->  PassSenke;

END OF  Senke

# ------------------------------------------------------------------------------

BASIC COMPONENT  Linienbus

MOBILE SUBCOMPONENTS OF CLASS  Passagier, Bus

DEFINITIONS
   DIMENSION  NStat := 10

   TABULAR FUNCTION  Fahrzeit (INT --> REAL)
   ON  (  1,   2,   3, ... ,  10)
   --> (1.2, 1.8, 2.5, ... , 1.5)
```

```
DECLARATIONS
   PRIVATE LOCATION  Strecke (SET OF  Bus) :=  10 Bus

   ENTRY   LOCATION  ARRAY {1..NStat}  Eingang (SET OF Passagier)
   EXIT    LOCATION  ARRAY {1..NStat}  Ausgang (SET OF Passagier)

INITIAL CONDITIONS
   Strecke: Bus[1]. StatNo :=  1;         #  Aufenthaltssort zu Beginn
   Strecke: Bus[2]. StatNo :=  3;
   ...

DYNAMIC BEHAVIOUR

   FOREACH  B  IN  Strecke
   LET
       IF  T >= B.TStation
       LET
#          Aussteigen der Fahrgaeste
#          -------------------------
           FOREACH  P  IN  B.Plaetze  |  P.Ziel = B.StatNo
           LET
               P -> Ausgang [B.StatNo];

               P.Ort^ :=  B.StatNo;
           END

#          Einsteigen der Fahrgaeste
#          -------------------------
           FOREACH  P  IN  Eingang [B.StatNo]
           LET
               P -> B.Plaetze;
           END

#          Naechster Halt
#          --------------
           B.TStation^ :=  T + FahrZeit (B.StatNo);
           B.StatNo^   :=  MOD (B.StatNo, NStat) + 1;
       END
   END

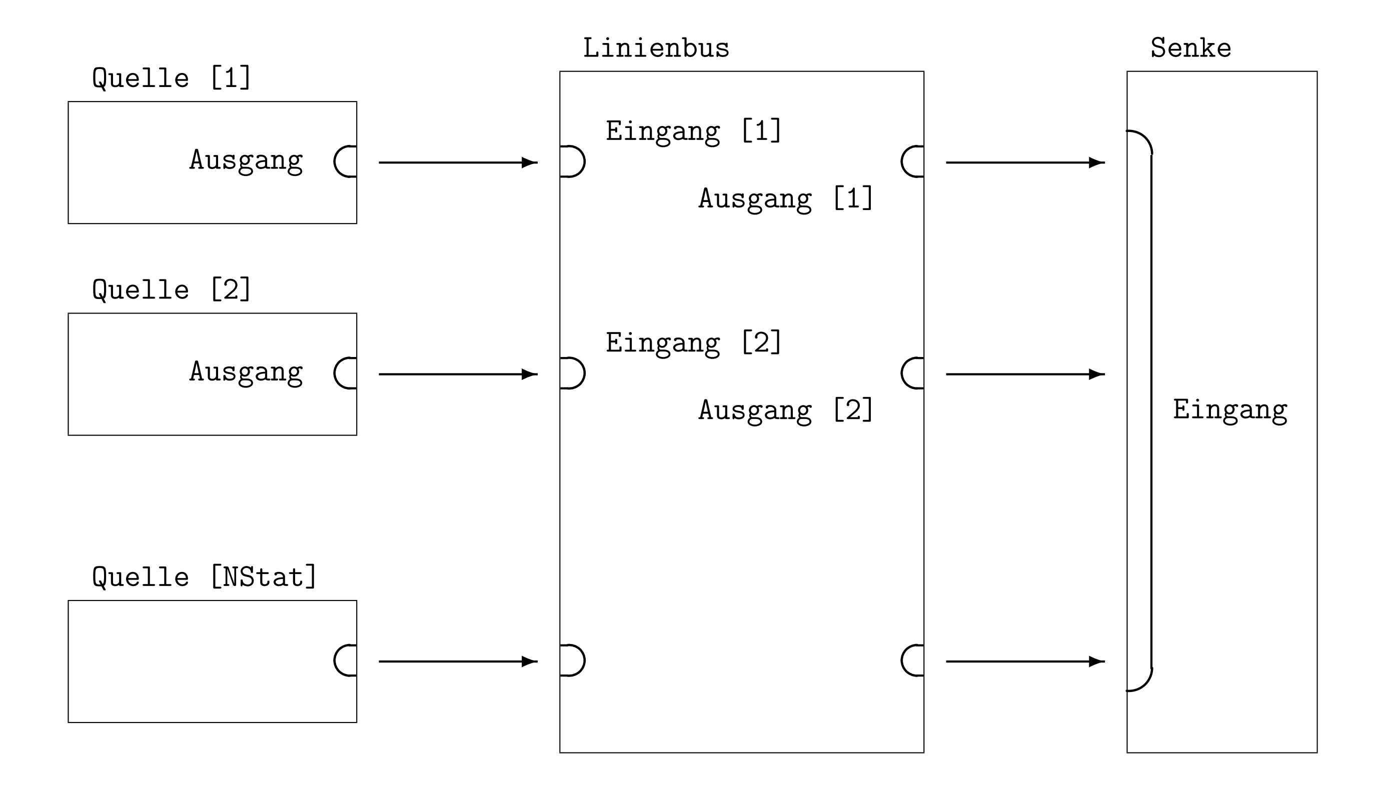
END OF  Linienbus
```

```
HIGH LEVEL COMPONENT  System                 # Nur zu Testzwecken

SUBCOMPONENTS   Linienbus,
                ARRAY {1..NStat}  Quelle,
                Senke

STRUCTURE
   TRANSPORT CONNECTIONS
   Quelle {i}. Ausgang      ==>  Linienbus. Eingang [i];
   Linienbus.  Ausgang {i}  ==>  Senke;

END OF  System
```



# Kapitel 8

# Zusammenfassung

In der vorliegenden Arbeit wurde ein Konzept für eine formale Sprache vorgestellt und diskutiert, das der Beschreibung und Simulation von dynamischen Modellen dient. Dabei lag der Schwerpunkt auf Warteschlangen- und Transportmodellen.

## Die Problemstellung

Auftrags-orientierte Systeme wie Rechnernetze, Betriebsorganisationen und Betriebssysteme sowie Transport- und Verkehrssysteme zeichnen sich dadurch aus, daß ihre Leistungsfähigkeit in erheblichem Maß von den eingesetzten Steuerungen bestimmt werden. Will man die Leistungsfähigkeit eines solchen Systems anhand von Leistungsgrößen bestimmen und zu diesem Zweck das System simulieren, dann muß das Simulationsprogramm die wesentlichen Teile der Steuerungssoftware enthalten. Da die klassischen Simulationsssprachen hierzu nicht über die nötige Funktionalität verfügen und datengesteuerte Baustein-Simulatoren nicht mit individuellen Steuerungskomponenten versehen werden können, konnten derartige Systeme nur in einer höheren Programmiersprache - evtl. unter Zuhilfenahme eines Simulationspakets modelliert werden. Wegen der Komplexität der Programme kamen hierfür eigentlich nur Informatiker in Frage.

Simulationsprogramme dieser Art benötigen jedoch einen beträchtlichen Zeitaufwand für die Erstellung und sind dennoch sehr schwer lesbar und änderungsunfreundlich. Da einerseits der Informatiker meist nur wenig von der Anwendung versteht und der Anwender das Simulationsprogramm nicht nachvollziehen kann, läßt sich nie sicherstellen, ob das Simulationsprogramm genau den Modellvorstellungen des Anwenders entspricht.

## Die Zielsetzung

Auftrags-orientierte Systeme bzw. Warteschlangen- und Transportsysteme sollten daher durch eine Sprache mit einem einfacheren Paradigma beschreibbar sein. Durch ein einfaches und schnell erlernbares Sprachkonzept und gut lesbare und verständliche Formulierungen soll dem Anwender die Möglichkeit gegeben werden, die Beschreibungen der Modelle zu verstehen. Die Formulierung des Modells sollte nicht länger einen Informatiker erforderlich machen. Ein geschulter Anwender oder ein Ingenieur sollte nach einer gewissen Einarbeitungszeit selbst in der Lage sein, Modelle zu erstellen oder zumindest bestehende Modelle zu modifizieren.

Aus dieser formalen Modellbeschreibung sollte es dann möglich sein, mittels eines Compilers ein effizient ausführbares Simulationsprogramm zu generieren, das mit den nötigen Schnittstellen versehen ist, um in einer Experimentierumgebung ablauffähig zu sein.

Da die Modellierung auftrags-orientierter Systeme hohe Anforderungen an die Funktionalität stellt, sollte eine universell einsetzbare Sprache, d.h. eine Sprache mit einem elementaren Abstraktionsniveau geschaffen werden. Dadurch sollte gleichzeitig ein breites Anwendungsfeld erschlossen und eine detailgetreue Modellierung sichergestellt werden.

Die Sprache sollte auch als Basis für Baustein-Simulatoren dienen können und einen graphischen Modellaufbau ermöglichen. Die wichtigste Voraussetzung hierfür war es, Modelle modular und hierarchisch aufbauen zu können.

## Der Lösungsweg

Von der in der Systemtheorie üblichen Modelldarstellung war bekannt, daß sie die genannten Anforderungen erfüllt und sich auch sehr gut für die Umsetzung in eine Computersprache eignet. Zudem ist es möglich, die kontinuierliche und die diskrete Modellierungswelt in einem einheitlichen Konzept unterzubringen. Dies zeigte sich bereits als Ergebnis meiner Dissertation [Esch 90].

In dieser Abhandlung wurde nun untersucht, ob dieses Paradigma, d.h. die explizite, zustandsorientierte Modelldarstellung, auch für die Beschreibung von Warteschlangen- und Transportsystemen taugt.

Zunächst wurde versucht, durch die Einführung von Elementen (Verbunde von Modellvariablen), von Feldern (indizierbaren Attributen und Elementen) und von Allquantoren das Problem zu erledigen. Es zeigte sich, daß hierdurch jedoch keine Modularisierung der Modelle möglich ist. Es wurden Ergänzungen notwendig, die das bisherige Konzept um grundlegend neue Inhalte erweiterten.

Neben den Modellvariablen als Repräsentanten für Attribute wurden Variablen als Repräsentanten für (geordnete) Mengen eingeführt, wobei jedes Mengenelement selbst wiederum einen Verbund von Variablen repräsentiert. Es erwies sich als besonders vorteilhaft, diese Mengen als disjunkt anzusehen. Dadurch erhielten sie die Bedeutung eines Aufenthaltsorts für Elemente und wurden daher auch als Bestandsgrößen oder Locations bezeichnet.

Um das dynamische Verhalten der Bestände zu beschreiben, wurde für diesen neuen Variablentyp eine Transportoperation eingeführt. Diese diente nicht nur zum Transportieren, sondern auch zum Erzeugen und Vernichten von Mengenelementen.

Es war das Ziel, eine Notation zu finden, die Transporte von unterscheidbaren und von nicht unterscheidbaren Elementen in gleicher Weise beschreiben. Zu diesem Zweck wurden auch Bestandsgrößen für Mengen eingeführt, deren Elemente sich nicht unterscheiden lassen und die lediglich durch eine Anzahl repräsentiert werden.

Es zeigte sich, daß es bei Modellen ohne unterscheidbare Elemente nur deshalb nicht üblich ist, einen Unterschied zwischen Attributen und Beständen zu machen, weil die einzelnen Elemente nicht individuell verfolgt werden müssen und ein Transport daher in zwei Operationen (Erzeugen und Vernichten) zerlegt werden kann.

Die Verwendung der Transportoperation hat neben einer größeren Anschaulichkeit jedoch den weiteren Vorteil, daß die Einhaltung des Erhaltungssatzes garantiert wird. Der Erhaltungssatz gilt naturgemäß für alle Bestandsgrößen.

Es wurde ausführlich diskutiert, in welcher Weise Modelle mit Bestandsgrößen modularisiert werden können. Da die Transportoperation jeweils zwei Bestände verändert, muß man sich entscheiden, in welcher Komponente die Transportoperation unterzubringen ist. Einer der beiden Bestände ist daher auf zwei Komponenten zu verteilen.

Da ein Bestand im Gegensatz zu einem Attribut nicht mehr nur einer Komponente zugerechnet werden kann, besitzt eine Komponente keine volle Autonomie bezüglich dieser verteilten Bestände.

Die Forderung nach modularer Modellzerlegung und die Forderung nach einem eindeutig geregelten Zugriff auf die Elemente und deren Attribute lassen sich aber nur gleichzeitig erfüllen, wenn die Komponenten nicht zuviel von ihrer Autonomie abgeben. Da die Bedürfnisse von Anwendung zu Anwendung unterschiedlich liegen, wurde eine Kompromißlösung vorgeschlagen, die es dem Anwender ermöglicht, bei der Deklaration verteilter Bestände selbst Schwerpunkte zu setzen und die Autonomie der Komponenten je nach Bedarf einzuschränken.

# Die Ergebnisse

Durch Einbeziehung aller erforderlichen Gesichtspunkte entstand nach und nach ein vollständiges Sprachkonzept. Die mit diesem Konzept erzielten Ergebnisse lassen sich wie folgt zusammenfassen:

Die konzipierte Modellbeschreibungssprache stellt eine Spezifikationssprache dar. Sie ist deklarativ und beschreibt die Modelldynamik als eine Menge von Aussageformen. Im Gegensatz zu einer prozeduralen Sprache formuliert sie lediglich *was* zu tun ist, um ein Modell abzuarbeiten, aber nicht *wie* die Ausführung erfolgen soll. Der Anwender muß sich demnach nicht um den Programmfluß kümmern, was insbesondere bei parallelen Abläufen eine wesentliche Erleichterung darstellt. Ein charakteristisches Merkmal für den deklarativen Charakter ist es, daß die Aussageformen in keiner besonderen Reihenfolge angeordnet werden müssen.

So konnte eine klare Unterscheidung zwischen Modellbeschreibung und Modellimplementation hergestellt werden. Während die Modellbeschreibung die physikalische Realisierung des Modells völlig offen läßt, ist die Modellimplementation in Form eines Simulationsprogramms immer an eine (bestimmte Art von) Rechenanlage gebunden.

Die vorgestellte Sprache besitzt ein sehr niedriges Abstraktionsniveau. Für alle Arten von diskreten Modellen sowie für kontinuierliche Modelle, die sich mit gewöhnlichen Differentialgleichungen beschreiben lassen, ist sie als universell anzusehen. Modelle können einen beliebigen Detaillierungsgrad erreichen. Für die allermeisten Anwendungen kann sie daher tatsächlich alternativ zu einer höheren Programmiersprache eingesetzt werden.

Auch wenn spezialisierte Werkzeuge leichter zu erlernen und zu handhaben sind, bietet sie gegenüber diesen den entscheidenden Vorteil, daß nicht für jede Anwendung ein anderes Werkzeug verwendet werden muß und daß Modelle verschiedener Anwendungsbereiche miteinander verknüpft werden können.

Abgesehen vom IF- und WHEN- Konstrukt ist die Sprache nicht redundant, d.h. sie besitzt keine überzähligen Konstruktionen.

Die Semantik der Sprache ist mit der zustandsorientierten Betrachtungsweise an ein verhältnismäßig einfaches Paradigma gebunden, das in den Ingenieurwissenschaften ohnehin weite Verbreitung genießt und daher den meisten Anwendern bereits vertraut ist.

Die Korrektheit einer Modellbeschreibung läßt sich weitestgehend bereits vom Compiler prüfen, so daß zur Laufzeit nur noch sehr wenige Fehlermöglichkeiten bestehen. Durch die Überprüfung auf Eindeutigkeit und Vollständigkeit (und damit auch auf Widerspruchsfreiheit) wird dem Anwender der deklarative Sprachcharakter sehr gut vor Augen geführt. Das ist vor allem deshalb erforderlich, weil sich die Notation nicht besonders von der prozeduraler Sprachen wie PASCAL unterscheidet.

Aus einer derartigen Modellbeschreibung läßt sich Code generieren, der eine (fast) optimale Laufzeiteffizienz besitzt. Es gibt also keinen Grund, ein Modell nur deshalb in einer prozeduralen Sprache abzufassen, weil es sich schneller ausführen ließe.

Modelle sind modularisierbar und können hierarchisch in Komponenten zerlegt werden. Die Modularisierbarkeit ermöglicht zudem eine Modellbeschreibung auf zwei verschiedenen Abstraktionsniveaus. Sieht man von den mobilen Komponenten ab, dann wird auf der untersten Hierarchie-Ebene die Modelldynamik beschrieben und auf den darüberliegenden Ebenen die Modellstruktur.

Liegen, wie in vielen Fällen, alle benötigten Basiskomponenten bereits vor, so ist es ausreichend, lediglich die Modellstruktur anzugeben. Die Formulierung eines Modells ist dann besonders einfach und vor allem auch graphisch darstellbar.

Wegen des Klassenkonzepts bestehen auch besonders umfangreiche Modelle in der Regel nur aus wenigen verschiedenen Modellkomponenten. Sind diese sorgfältig getestet, dann können hierarchisch zusammengesetzte Modelle im Grunde kein fehlerhaftes Verhalten zeigen, es sei denn, die Verbindungen wurden nicht richtig vereinbart. Man erhält daher auch für große Modelle ein hohes Maß an Zuverlässigkeit.

Das Konzept der modularen Zerlegung ermöglicht auch den Aufbau von Modellbanken. Darunter sind Sammlungen von Modellkomponenten zu verstehen, die in einem bestimmten Anwendungsgebiet immer wieder benötigt werden. Beim Rückgriff auf solche Modellbanken ist die Modellerstellung ebenso einfach wie bei den datengesteuerten Baustein-Simulatoren.

Beschränkt man sich auf eine spezielle Modellbank, ist es kein Problem, eine komfortable graphische Bedienoberfläche hierzu anzubieten. Will man hingegen dem Benutzer die Möglichkeit geben, selbst neue Modellkomponenten in die Bank aufzunehmen und ein graphisches Symbol hierfür zu erzeugen, steht man vor einer weit schwierigeren Aufgabe. Die Behandlung dieser Thematik ist Gegenstand gegenwärtiger Forschung.

## Die Erfahrungen mit der Realisierung

Das vorgestellte Sprachkonzept wurde in geringfügig anderer Form mit der Modellbeschreibungssprache SIMPLEX-MDL [Esch 91] realisiert. Diese Sprache ist Bestandteil des Simulationssystems SIMPLEX-II [Lang 89, Esch 90, Dörn 91], das seit 1989 institutsintern und seit 1991 auch bei externen Anwendern im Einsatz ist. Dadurch liegen bereits viele Erfahrungen im Umgang mit der Sprache vor.

Beim Einsatz in Hauptseminaren hat sich gezeigt, daß Studenten der Informatik ohne weiteres Vorwissen in der Lage sind, innerhalb von ein bis zwei Wochen größere Modelle (10 Seiten Code) in SIMPLEX-MDL zu formulieren.

Es zeigte sich allerdings auch, daß der Anwender erst lernen muß, die neue Sprache effizient zu nutzen. Hier sind vor allem zwei Punkte zu nennen:

1. Der Anwender muß lernen, ein gegebenes System (z.B. einen Aufzug) auf die für die Fragestellung wesentlichen Teile zu reduzieren. Nur eine strenge top-down-Vorgehensweise kann hier zum Erfolg führen. Ausgehend von den interessierenden Größen sind nach und nach alle wesentlichen Einflüsse zu erfassen.

2. Der Anwender muß lernen, sein Modell zustandsorientiert zu betrachten. Gerade diskrete Modelle werden von Leuten mit Programmierkenntnissen zu gerne als eine Folge von Aktionen mit Zwischenzuständen gesehen. Nicht Ablaufdiagramme, sondern Darstellungen wie Zustandsübergangsdiagramme, Abhängigkeitsgraphen oder System Dynamics Diagramme können eine gute Hilfestellungen geben.

Der erste Punkt betrifft ganz allgemein die Systemanalyse und steht unabhängig vom verwendeten Simulationswerkzeug. Dennoch tritt er hauptsächlich bei einem universellen Werkzeug zutage. Die herkömmlichen Simulationssprachen mit ihren Funktionsblöcken und noch viel mehr die Baustein-Simulatoren drängen dem Anwender eine bestimmte Sichtweise der Realität auf, weil er sein Modell aus diesen Komponenten aufbauen muß.

Dies erweist sich aber in vielen Fällen von großem Nachteil, ja es lenkt den Anwender sogar von den entscheidenden Zusammenhängen ab. So hat es sich herausgestellt, daß man über die Leistungsfähigkeit eines flexiblen Fertigungssystems keine vernünftigen Auskünfte erhält, wenn man es in Werkstücke, Paletten, Maschinen, Wechsler u.s.w., d.h. in sämtliche Hardware-Komponenten zerlegt. Entscheidend ist in erster Linie die Abwicklung der Aufträge, d.h. die Software-Komponenten des Systems.

Beim zweiten Punkt geht es darum, daß der Anwender eine geeignete Modellierungstechnik verwendet. Ähnlich wie ein Programmierer erst die "Strukturierte Programmierung" erlernen und einüben muß, um effizient zu programmieren, muß auch der Modellierer bestimmte Techniken nutzen. Es ist noch eine Aufgabe weiterer Forschung, graphische Darstellungsmethoden für die zustandsorientierte Modellbeschreibung in dem Sinne zu vervollkommnen, wie Struktogramme eine Hilfestellung für "sauberes" Programmieren abgeben.

Aus diesen Gründen ist es wichtig, daß Systemanalyse und Modellierungstechniken gleichwertige Ausbildungsziele neben dem reinen Erlernen der Sprache (Syntax und Semantik) sind, damit der Anwender erfolgreich davon Gebrauch machen kann und die Modelle gut lesbar und übersichtlich werden. Ebenso wie man in einer prozeduralen Sprache unverständlich programmieren kann, ist es leider auch in einer Spezifikationssprache möglich, unverständlich zu formulieren.

Durch zahlreiche Anwendungen wurde daher versucht, einen geeigneten Modellierungsstil zu finden. Die Broschüre "Öffentliche Modellbanken des Simulationssystems SIMPLEX II" [ApEsWi 92] stellt viele beispielhafte Anwendungen vor und liefert den MDL-Quelltext dazu.

## Resumee

Mit der Modellbeschreibungssprache als universellem Werkzeug ist es sicherlich gelungen, das Erstellen von Simulationsprogrammen ganz erheblich zu vereinfachen. Da bewußt auf "schwierige" Sprachkonstrukte, wie Zeiger, die eine hohe Fehleranfälligkeit beinhalten, verzichtet wurde, ist es auch dem Nicht-Informatiker möglich, Modelle in der Sprache zu formulieren.

Andererseits ist es sicher auch gelungen, dem Anwender eine Modellbeschreibung als Dokumentation zu überlassen, die dieser auch versteht und in der er seine Modellvorstellung wiederfindet. Damit ist ein Anwender erstmals in der Lage, mit anderen Fachkollegen über ein Modell zu diskutieren, ohne daß irgendwelche Angaben fehlen oder unklar bleiben. Die Modellbeschreibung ist stets vollständig und eindeutig und enthält nicht mehr und nicht weniger Information als erforderlich.

Es bleibt die Frage, wer denn die Modelle formuliert. Ohne eine gewisse Ausbildung ist dies sicherlich nicht möglich. Idealerweise würde ein mathematisch vorgebildeter Anwender eine Zusatzausbildung erhalten und seine Modelle selbst formulieren. Dies wird er aber nur tun, wenn er selber Systemanalyse und Modellerstellung betreibt. Solche Anwender sind in der Regel nur im Hochschulbereich zu finden.

Der andere Typ von Anwender, der nur gelegentlich Simulationen durchführt, benötigt einen Partner, der nach seinen Angaben das Modell formuliert. Als Partner könnte man sich einen Simulationstechniker vorstellen, der Sprache und Methoden beherrscht und in der Lage ist, sie in unterschiedlichen Fachgebieten einzusetzen. Simulationstechniker könnten in Ingenieurbüros, Unternehmensberatungen oder in Planungsabteilungen größerer Firmen zu finden sein. Da der Anwender die Modellbeschreibung zumindest lesen kann, ist er auch in der Lage, die Arbeit des Simulationstechnikers nachzuvollziehen und zu überprüfen.

An dieser erforderlichen Zusatzqualifikation leidet bislang die Akzeptanz der Sprache. Da der Anwender meint, zum Erlernen der Sprache keine Zeit zu haben, und ein Simulationstechniker nicht so leicht zu finden ist wie ein PASCAL-Programmierer, bleibt er zunächst einmal bei den Werkzeugen, die er kennt und bereits im Einsatz hat.

Dabei sind die Vorteile gegenüber herkömmlichen Werkzeugen ähnlich wie der Übergang von einer Assemblersprache zu einer höheren Programmiersprache: Vergleicht man SIMPLEX-MDL mit dem ebenfalls universell einsetzbaren Simulationspaket GPSS-FORTRAN, einer FORTRAN-Implementation der Simulationssprache GPSS, dann läßt sich feststellen, daß man zur Beschreibung eines kleinen Modells (10 Zustandsübergänge) etwa ein Fünftel der Zeit und zum Test des Modells etwa ein Zehntel der Zeit benötigt. Mit wachsender Modellgröße nimmt der Unterschied rapide zu.

Diese Fakten allein sind für den Anwender jedoch nicht überzeugend und deshalb gilt es genauer zu untersuchen, was zu tun ist, um die Akzeptanz zu verbessern. Vorschläge hierzu gibt es genügend: Für jedes Anwendungsgebiet wird ein realistisches Beispiel benötigt, das als Vorbild weitere Modelle dienen kann; die Sprache und die Modellierungstechniken müssen im Selbststudium erlernbar sein; der Umgang mit dem System muß den Eindruck vermitteln, als wüßte man ohne nachzulesen, wie es zu bedienen ist; u.s.w. .

Trotz oder gerade wegen der vielen Bemühungen gewinnt man im Lauf der Zeit leider den Eindruck, daß die Qualität nicht der ausschlaggebende Gesichtspunkt für die Akzeptanz eines Erzeugnisses ist. Es ist schade, wenn deshalb so viele Entwicklungen an Hochschulen (und auch anderswo) im Sande verlaufen.

Für zukünftige Entwicklungen erscheint vor allem eine Verbreiterung des Anwendungsfeldes interessant. Die vorgestellte Spezifikationssprache wurde zwar zum Zweck der Simulation entworfen, aber auch im Bereich der Prozeßautomatisierung und der Steuerungstechnik könnte sie – evtl. in modifizierter Form – Verwendung finden.

In diesen Gebieten werden zur Durchführung von Simulationsstudien Modelle erstellt, die bereits große Teile des zu realisierenden Systems beinhalten. Es erscheint daher überflüssig, die Teile nochmals in einer Implementierungssprache zu codieren. Daher gilt es zu prüfen, ob es nicht möglich ist, in diesen Anwendungsgebieten auf eine Programmierung in einer Implementierungssprache vollständig zu verzichten und nur noch mit einer Spezifikationssprache zu arbeiten.

# Literaturverzeichnis


[ApEsWi 92] Apsel, Th.; Eschenbacher, P.; Wittmann, J.:
Öffentliche Modellbanken des Simulationssystems SIMPLEX II;
erschienen im Selbstverlag:
Institut für Mathematische Maschinen und Datenverarbeitung
der Universität Erlangen-Nürnberg, Erlangen 1992

[Böh 81] Böhme, Gert:
Einstieg in die Mathematische Logik,
Carl Hanser Verlag, München 1981

[BORIS 85] Siemens AG (Hrsg.):
Benutzerhandbuch, BORIS-Version 1.5

[Boss 85] Bossel, Hartmut:
Dynamik des Waldsterbens,
Springer Verlag, Fachberichte Simulation Bd. 4, Berlin 1985

[BraFo 87] Bratley, P.; Fox, B.; Schrage, L.:
A Guide to Simulation,
Springer Verlag, New York 1987

[Cell 83] Cellier, Francois:
Simulation Software: Today and Tomorrow
in: Simulation in Engineering Sciences,
IMACS Symposium International, Nantes 1983

[ChaMi 88] Chandy, K.M.; Misra, J.:
Parallel Program Design,
Addison-Wesley, Reading, Massachusetts 1988

[Crae 85] Craemer, Diether:
Mathematisches Modellieren dynamischer Vorgänge;
Eine Einführung in die Programmiersprache DYNAMO,
Teubner Verlag, Stuttgart 1985

[Dörn 91] Dörnhöfer, Klaus:
Die Benutzerumgebung von SIMPLEX II - Eine offene graphische
Bedienoberfläche zur Erstellung und Auswertung von Simulationsmodellen,
Arbeitsberichte des Instituts für Mathematische Maschinen und
Datenverarbeitung der Universität Erlangen-Nürnberg,
Dissertation, Band 24, Nummer 8, Erlangen 1991

[Esch 90] Eschenbacher, Peter:
Entwurf und Implementierung einer formalen Sprache zur Beschreibung dynamischer Modelle,
Arbeitsberichte des Instituts für Mathematische Maschinen und Datenverarbeitung der Universität Erlangen-Nürnberg,
Dissertation, Band 23, Nummer 1, Erlangen 1990

[Esch 91] Eschenbacher, Peter:
Die Modellbeschreibungssprache SIMPLEX-MDL,
in: Referenzhandbuch des Simulationssystems SIMPLEX II,
erschienen im Selbstverlag, (c) SIEMENS AG,
Institut für Mathematische Maschinen und Datenverarbeitung der Universität Erlangen-Nürnberg, Erlangen 1991

[EsWi 90] Eschenbacher, P.; Wittmann, J.:
Methodische Vorgehensweise bei der Durchführung von Simulationsstudien,
Skript (Teil B) zur Vorlesung Šimulation IIïm WS 89/90

[Forr 72a] Forrester, Jay W.:
Grundzüge einer Systemtheorie
Gabler Verlag, Wiesbaden 1972

[Forr 72b] Forrester, Jay W.:
Der teuflische Regelkreis
Deutsche Verlagsanstalt, Stuttgart 1972

[Forr 73] Forrester, Jay W.:
World Dynamics,
Wright-Allen Press, Cambridge, MA, 1973

[INGER 85] INGERSOLL Engineers (Hrsg.):
Flexible Fertigungssysteme: Der FFS-Report der INGERSOLL ENGINEERS,
Springer Verlag, Berlin 1985

[Kaehl 86] Kaehler, T.; Patterson, D.:
A Taste of Smalltalk,
Norton, New York 1986

[KerRit 83] Kernighan, B.; Ritchie, D.:
Programmieren in C,
Carl Hanser Verlag, München 1983

[Kett 83] Kettenis, D.L.:
The COSMOS Modelling and Simulation Language,
in: Ameling, W. (Hrsg.):
Proceedings of the First European Simulation Congress 1983,
Springer, Berlin 1983, pp 251-260

[Kett 88] Kettenis, D.L.:
The COSMOS Simulation Environment
in: Huntsinger, R.C., SCS International (Hrsg.):
Simulation Environments,
Proceedings of the ESM, Nice 1988,

[KöMa 72] Köcher, D.; Matt, G.; Oertel, C.; Schneeweiß, H.:
Einführung in die Simulationstechnik,
Hrsg.: Deutsche Gesellschaft für Operations Research (DGOR),
Beuth Vertrieb, Berlin 1972

[Lang 89] Langer, Klaus-Jürgen:
Entwurf und Implementierung eines Simulationssystems mit integrierter Modellbank- und Experimentdatenverwaltung
Arbeitsberichte des Instituts für Mathematische Maschinen und Datenverarbeitung der Universität Erlangen-Nürnberg,
Dissertation, Band 22, Nummer 17, Erlangen 1989

[MarLed 87] Marcotty, M.; Ledgard, H.:
The World of Programming Languages,
Springer Verlag, Berlin 1987

[May 89] Mayer, Klaus-Peter:
Modellierung der Schadstoffeinwirkung auf das Ökosystem Wald,
Studienarbeit am IMMD4 der Universität Erlangen, 1989

[Ören 84] Ören, Tuncer I.:
GEST - A Modelling and Simulation Languages Based on System Theoretic Concepts,
in: Ören, T.I. (Hrsg.):
Simulation and Model-Based Methodologies: An Integrative View,
NATO ASI Series, vol. F10, pp 281-334
Springer Verlag, Berlin 1984

[ÖrZi 79] Ören, T.; Ziegler, B.:
Concepts for Advanced Simulation Methodologies,
in: SIMULATION, Volume 32, No. 3, March 1979, pp 69-82

[Page 88] Page, B.; Bölckow, R.; Heydermann, A.; Kadler, R.; Liebert, H.:
Simulation und moderne Programmiersprachen,
Springer Verlag, Fachberichte Simulation Bd. 8, Berlin 1988

[Ped 86] Pedgen, C. Dennis:
Introduction to SIMAN,
Systems Modelling Corporation, Calder Square, PN, 1986

[Pich 75] Pichler, Franz:
Mathematische Systemtheorie, Dynamische Konstruktionen,
Walther de Gruyter, Berlin 1975

[Prit 74] Pritsker, A. Alan B.:
The GASP IV Simulation Language,
John Wiley, New York 1974

[PriPed 79] Pritsker, A. Alan B.; Pedgen, C. D:
Introduction to Simulation and SLAM,
John Wiley, New York 1979

[Prit 82] Pritsker & Associates (Hrsg.):
A Comparison of three general purpose Simulation Languages:
SLAM, SIMSCRIPT and GPSS,
erschienen im Selbstverlag:
P.O. Box 2413, West Lafayette, Indiana 47906; Oktober 1982

[Richt 93] Richter, Oliver:
Statistische Disposition in flexiblen Fertigungssystemen,
Diplomarbeit am IMMD 4 der Universität Erlangen, 1993

[Rohl 73] Rohlfing, Helmut:
SIMULA
Bibliographisches Institut, Mannheim 1973

[Russ 83] Russel, Edward C.:
Building Simulation Models with SIMSCRIPT II.5,
CACI, Los Angeles 1983

[Schm 84a] Schmidt, Bernd:
Der Simulator GPSS-FORTRAN Version 3,
Springer Verlag, Fachberichte Simulation Bd. 2, Berlin 1984

[Schm 84b] Schmidt, Bernd:
Modellbildung mit GPSS-FORTRAN Version3,
Springer Verlag, Fachberichte Simulation Bd. 3, Berlin 1984

[Schm 85] Schmidt, Bernd:
Systemanalyse und Modellaufbau - Grundlagen der Simulationstechnik,
Springer Verlag, Fachberichte Simulation Bd. 1, Berlin 1985

[Schm 88] Schmidt, Bernd:
Simulation von Produktionssystemen
in: Simulation in der Fertigungstechnik,
Springer Verlag, Fachberichte Simulation Bd. 10, Berlin 1988

[Schnu 91] Schnupp, Peter (Hrsg.):
Moderne Programmiersprachen,
Oldenbourg Verlag, München 1991

[Schr 91] Schriber, Thomas J.:
An Introduction to Simulation Using GPSS/H,
John Wiley & Sons, New York 1991

[Schür 77] Schürmann, Hans-Werner:
Theoriebildung und Modellbildung,
Akademische Verlagsgesellschaft, Wiesbaden 1977

[Sieg 91] Siegert, Hans Jürgen:
Simulation zeitdiskreter Systeme,
Oldenbourg Verlag, München 1991

[Strou 87] Stroustrub, Bjarne:
The C++ Programming Language,
Addison Wesley, Reading, MA, 1987

[SYSMOD 86] System Designers (Hrsg.):
SYSMOD User Manual, Release 1.0, April 1986
SYSMOD Language Definition,Release 1.0, Juni 1986,
erschienen im Selbstverlag:
System Designers Scientific, Ferneberga House, Alexandra Road,
Farnborough, Hampshire, United Kingdom

[Ulb 91] Ulbrich, Alwin:
Modellierung und Simulation einer Werkstattfertigung,
Studienarbeit am IMMD4 der Universität Erlangen, 1991

[Ulb 92] Ulbrich, Alwin:
Definition und Implementierung der Schnittstelle zwischen
Simulationsprogramm und Experimentierumgebung in SIMPLEX III,
Diplomarbeit am IMMD4 der Universität Erlangen, 1992

[Unbe 80] Unbehauen, Rolf:
Systemtheorie - Eine Darstellung für Ingenieure,
Oldenbourg Verlag, München 1980

[Wun 86] Wunsch, Gerhard:
Handbuch der Systemtheorie,
Oldenbourg Verlag, München 1986

[ZeiElz 79] Ziegler, B.; Elzas, M.; Klir; Ören, T. (Hrsg.):
Methodology in Systems Modelling and Simulation,
North Holland, Amsterdam 1979